Scalable hosting of web applications

Sivasubramanian, S.

2009

document version
Publisher's PDF, also known as Version of record

SCALABLE HOSTING OF WEB APPLICATIONS

SWAMINATHAN SIVASUBRAMANIAN

VRIJE UNIVERSITEIT

SCALABLE HOSTING OF WEB APPLICATIONS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op woensdag 18 april 2007 om 10.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

SWAMINATHAN SIVASUBRAMANIAN

geboren te Chennai, India

promotoren: prof.dr.ir. M.R. van Steen
dr.ir. G.E.O. Pierre

ACKNOWLEDGEMENTS

My grad school advisor once remarked that a Ph.D. degree is not a testimonial of one's intellect but his/her perseverance. A few people have helped me through this journey and it is time to thank them!

A major share of the credit goes to my advisors: Maarten van Steen and Guillaume Pierre. Maarten has always been an energetic advisor providing excellent support, motivation and guidance. He has taught me a great deal about distributed systems, scalability, and also authoring. Moreover, he has been a constant source of motivation and really raised the quality of my research. He introduced me to the field of CDNs and I owe a great deal of my current knowledge and skills to him. Similarly, Guillaume has been a great advisor. He has always been accessible and never objected in me barging into his office to discuss any of my wacky (and most often lousy) ideas. He has been remarkably patient in reading the first drafts of my papers (a.k.a. brain dumps) and making them more readable. I owe a great deal to both, Maarten and Guillaume, for playing a crucial rule in my research, mentality and success.

Guillaume has also been more than just an advisor, he has been a great friend and helped me get through the initial dutch culture shock. I have to thank him and Caroline for being kind enough to treat me and my group mates to some great french cuisine every semester!

During my Ph.D. research, I was fortunate to work with some great people. Among them, first, I would like to thank Gustavo Alonso for helping me with my research. I knew nothing about databases and my visit to ETH helped a great deal in shaping my thesis research. I would also like to thank Bruno Crispo for introducing me to the field of security that helped me in generating lots of publications in that field. Similarly, I also need to thank Sandjai Bhulai for exposing me to the field of queueing theory that enabled us to generate some good results in the field of multi-tier application modeling.

I would also like to thank Werner Vogels for providing me the opportunity and encouraging me to take on some real world problems in distributed systems scalability. His thoughts on scalability and distributed systems have changed my research perspective. Also, I would also like to the other members in my thesis

committee, Michael Rabinovich, Andy Tanenbaum, and Rob van der Mei, for their effort and time in reading reviewing this dissertation.

Now, to my colleagues at the Vrije Universiteit. First, I owe a great deal to my P4.30 office mates: Michal and Spyros. These guys made my four years VU stay more fun with some intriguing conversations on various topics ranging from research, iraq war, spirituality, amsterdam and so on... Many thanks to Arno for writing the dutch summary and helping me with the thesis formatting! Also, I would like to thank all the members in compsys including Bogdan, Jan Mark, Arno, Berry, Srijith, Elth, Wilfred, Konrad, Vivek, Daniela and Melanie for making workplace fun.

Now, to my friends in Amsterdam: Narayanan, Paul, Jai, Abhilash, Felix, Prabhakar, Madhav, and Anita. I cannot thank these people enough who made my life in Amsterdam liveable. Especially, I owe a lot to Narayanan for organizing some great road trips around Europe. Many thanks to Jai - his immense enthusiasm for learning distributed systems forced me to keep my skills sharp to take his constant flow of questions everyday.

Last but certainly not the least, I owe this thesis to my family: my dad, mom, brother and manni. My dad and mom have always been a constant source of support and love which helped me get through the four years easily. Especially, I owe my Ph.D. to my brother, Srinivas, as he was the one who took care of my family financial aspects while I had the luxury of being an irresponsible student.

Swaminathan Sivasubramanian
Seattle, USA. Feb. 2007.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

The World Wide Web ("WWW" or simply the "Web") is a global information medium which users can access via computers connected to the Internet.[1] The Web, which began as a simple networked information project at a research lab, has grown to be the most popular medium of digital communication. It has become an integral point of people's lives and several day-to-day activities such as shopping, banking and trading can now be performed through the Web.

The growing popularity of the Web has driven the need for several businesses such as online news, retail and financial, to open their business processes to their Web clients. For online businesses, providing a good client experience is one of their primary concerns. Human computer interaction studies have shown that frequent users prefer response times of less than a second for most tasks, and that human productivity improves more than linearly as computer systems response times fall in the sub-second range [Shneiderman, 1984]. Hence, for such online businesses, providing low response latency to their clients is a crucial business requirement [Vogels, 2006].

The latency incurred in receiving a response from a Web site can be split into two components. The first component is due to the latency incurred by the request and the response to traverse the network between the client and the server machines. The second component is the page generation latency, i.e., the time incurred by the Web site to generate the appropriate response to each client.

A popular technique used by leading content providers to reduce their site's response latency is to avail their digital content through content delivery networks (popularly known as CDNs) [Rabinovich and Spastscheck, 2002; Sivasubramanian et al., 2004b]. CDNs like Akamai [Dilley et al., 2002] and Speedera[2] deploy servers (called edge servers) around the Internet that locally cache Web content

[1] http://en.wikipedia.org/wiki/World_Wide_Web

[2] http://www.speedera.com

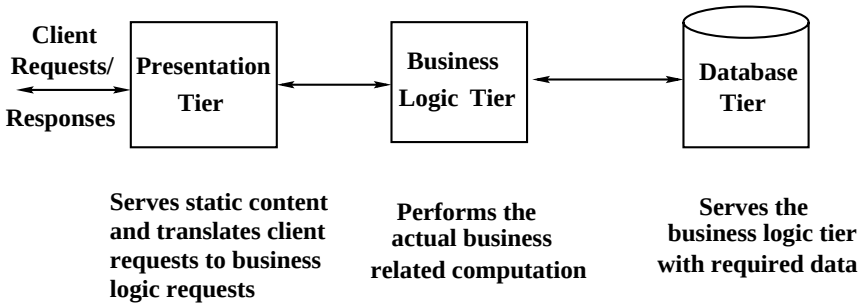


Figure 1.1: Application model of a simple 3-tier Web application.

and deliver them to the clients. By serving Web content from servers located close to the clients, CDNs reduce the network latency component as each request need not travel across a wide-area network. Various commercial CDNs (e.g., Akamai, RADAR [Rabinovich and Aggarwal, 1999], and Speedera) are successful in building scalable infrastructure to host numerous popular Web sites in the Internet. These CDNs are usually suited for hosting static Web content, i.e., content that does not get updated too often. Examples of static Web contents include images, multimedia files, and rarely updated HTML pages.

However, modern Web sites such as Amazon.com and yahoo.com do not simply deliver static pages but generate content on the fly each time a request is received, so that the pages can be customized for each user. Generating a Web page in response to every request takes more time than simply fetching static HTML pages from disk. This is because these responses are produced by Web applications that issue one or more queries to database(s). Hence, the problem of scalable hosting of dynamic Web sites translates to scalable hosting of Web applications.

The application model of a simple Web application is given in Figure 1.1. As seen in the figure, a Web application usually consists of three tiers: presentation, business logic and database. The presentation tier deals with serving static content such as images and multimedia related content and is also responsible for translating an incoming client request to appropriate business logic request. The business logic tier is the heart of the application and contains the core business functionality. It receives requests from its preceding tier and generate appropriate responses by issuing one or more queries to the underlying database tier.

This thesis addresses the following research question: *How to host a multi-tiered Web application in a scalable and efficient manner?* In contrast to static Web sites, each HTTP request to a dynamic Web site can lead to one or more business logic requests, and multiple database queries. The most straightforward technique used by CDNs to improve the performance of such Web applications is Web page caching where (fragments of) the HTML pages generated by the application are cached for serving future requests [Challenger et al., 2005]. This

technique works well only if many requests to the Web site can be answered with the same cached HTML page. However, with the growing drive towards personalization of Web sites, generated pages tend to be unique for every user, thereby reducing the benefits of conventional page caching techniques. Furthermore, if the underlying database gets updated often then the cached pages have to be evicted often to ensure that the clients are not delivered stale responses.

The limitations of page caching techniques have triggered the research community into investigating new approaches for scalable hosting of Web applications. To scale a Web application, one has to scale each of its tiers appropriately. Scaling up the presentation layer allows Web applications to scale static content delivery and involves running multiple instances of Web servers (possibly around the Internet). As we show in Chapter 2, the techniques used to scale static content delivery are well understood. Similarly, scaling up the business-logic tier involves replication of the application code across multiple servers. This problem is also well understood provided that the application code is stateless [Rabinovich et al., 2003; Cao et al., 1998; Seltzsam et al., 2006].

Scaling up the database tier is harder and has a significant impact on the performance of database-driven Web applications. For such applications, mere replication of the application code at the edge servers and a centralized database system may not suffice, as generation of each page may still require the application code to make many queries to the database. In such cases, having a central database server will increase the access latency to the database (as each database query is transmitted across the wide-area network), thereby resulting in high response latency. On the other hand, a simple strategy of replicating the entire database at all edge servers may result in huge performance overhead due to consistency maintenance.

THESIS CONTRIBUTIONS AND OUTLINE

In this thesis, we present several techniques that can be used to improve the performance of Web applications in an Internet wide CDN environment. In particular, we focus on relational database driven Web applications. Examples of relational database management systems include IBM DB2[3], Oracle[4], MySQL[5] and PostgreSQL[6]. The rest of the thesis is structured as follows. In Chapter 2, we present related work. Moreover, we propose a framework that aids in analyz-

[3] http://www.ibm.com/db2

[4] http://www.oracle.com/database/index.html

[5] http://www.mysql.org

[6] http://www.postgresql.org

ing, comparing and understanding several research efforts conducted in the area of CDNs.

In Chapter 3, we present GlobeCBC, a content-blind query caching middleware. Unlike existing data caching middleware systems, GlobeCBC stores the query results independently and does not merge different query results. We study the potential performance of this approach using extensive experimentations on our prototype implementation and compare it with other systems over an emulated wide-area network. Our evaluations show that content-blind caching performs well in terms of client latency for applications that exhibit high query locality. Moreover, it allows the system to sustain higher throughput by offloading the database tier.

In Chapter 4, we present GlobeDB, a system that performs autonomic replication of application data. While GlobeCBC is suited mostly for applications with high query locality, applications that do not have these characteristics require replication of the application data to have low access latencies to the database. GlobeDB is primarily targeted for such applications and handles distribution and partial replication of application data *automatically* and *efficiently*. It provides Web applications the same advantages that CDNs offer to traditional static Web sites: low latency and reduced network usage. We substantiate these claims with extensive experimentation using a prototype implementation with an industry standard benchmark.

In Chapter 5, we present GlobeTP, a database replication technique that can be used to improve the throughput of the database tier. The motivation behind this technique is the observation that generic replication algorithms used by most databases do not scale linearly as they require to apply all update, insertion and deletion (UDI) queries to every database replica. The system throughput is therefore limited to the point where the number of UDI queries alone is sufficient to overload one server [Fitzpatrick, 2004]. In such scenarios, partial replication of a database can help, as update queries will be executed only by a subset of all servers. Even though GlobeDB employs partial replication, it does not support queries that span across multiple (partially replicated) data items (e.g., queries spanning multiple database tables). GlobeTP exploits the fact that a Web application's query workload is based on a small set of read and write templates. Using knowledge of these templates and the query execution costs of different templates, GlobeTP provides database table placements that produce significant improvements in database throughput. We demonstrate the efficiency of this technique using different industry standard benchmarks.

The aforementioned middleware and techniques aim to improve the performance of Web applications by improving the performance of the database tier. Moreover, there are several page caching and business logic replication techniques

that aim to improve the performance of the presentation and business-logic tier. All these techniques aim for application scalability by improving the performance of a single tier. However, from the view point of an administrator, he/she is interested in the end-to-end performance of the Web application and not just the performance of individual tiers. Hence, capacity provisioning of Web application also needs to be done based on its end-to-end performance.

To this end, in Chapter 6, we present a novel resource provisioning approach for multi-tiered Internet services. In contrast to previous works on capacity provisioning of Web systems, we propose to select the resource configuration based on its end-to-end performance instead of optimizing each tier individually. The proposed approach employs a combination of queueing models and runtime cache simulations. Our experiments with a wide range of application demonstrate that our approach, compared to optimizing tiers independently, is able to host services more scalably with less resources.

The aforementioned provisioning approach is based on a simple analytical model that allows us to compute only the mean end-to-end response time of an application. However, many e-commerce companies are not just interested in the mean response time but also in its variability [Vogels, 2006]. To this end, in Chapter 7, we present an analytical model for multi-tiered software systems and derive exact and approximate expressions for the mean and the variance, respectively, of the end-to-end response times. We validate our expressions through extensive experimentations with well-known industry-standard benchmarks and services for a wide range of resource configurations. Our experiments show that our model is highly accurate in estimating both the mean and the variance in response times with a margin of error less than 10 percent. Furthermore, we discuss and demonstrate the benefits of the model to resource provisioning, service level agreement (SLA) negotiation, and admission control. Finally, Chapter 8 concludes the thesis and discusses the open issues.

CHAPTER 2

Background and Related Work

In this chapter, we survey research efforts that address different aspects of building a scalable CDN. A version of this chapter have been published in [Sivasubramanian et al., 2004b].

2.1. INTRODUCTION

Replication is a technique that allows to improve the quality of distributed services. In the past few years, it has been increasingly applied to Web services, notably for hosting Web sites. In such cases, replication involves creating copies of a site's Web documents, and placing these document copies at well-chosen locations. In addition, various measures are taken to ensure (possibly different levels of) consistency when a replicated document is updated. Finally, effort is put into redirecting a client to a server hosting a document copy such that the client is optimally served. Replication can lead to reduced client latency and network traffic by redirecting client requests to a replica closest to that client. It can also improve the availability of the system, as the failure of one replica does not result in entire service outage.

These advantages motivate many Web content providers to offer their services using systems that use replication techniques. We refer to systems providing such hosting services as *replica hosting systems*. The design space for replica hosting systems is big and seemingly complex. In this chapter, we concentrate on organizing this design space and review several important research efforts concerning the development of Web replica hosting systems. A typical example of such a system is a Content Delivery Network (CDN) [Hull, 2002; Rabinovich and Spastscheck, 2002; Verma, 2002].

In this chapter, we survey a wide range of articles detailing the efforts carried

out in the area of building a scalable CDN. However, analyzing and comparing these efforts is difficult as these works address different aspects of Web replication. To this end, we propose a framework that aids us in understanding, analyzing and comparing the efforts conducted in this area. The framework covers the important issues that need to be addressed in the design of a Web replica hosting system. It is built around an *objective function* – a general method for evaluating the system performance. Using this objective function, we define the role of the different system components that address separate issues in building a replica hosting system.

The Web replica hosting systems we consider are scattered across a large geographical area, notably the Internet. When designing such a system, at least the following five issues need to be addressed:

1. How do we *select and estimate the metrics* for taking replication decisions?
2. *When* do we replicate a given Web document?
3. *Where* do we place the replicas of a given document?
4. How do we *ensure consistency* of all replicas of the same document?
5. How do we *route client requests* to appropriate replicas?

Each of these five issues is to a large extent independent from the others. Once grouped together, they address all the issues constituting a generalized framework of a Web replica hosting system. Given this framework, we compare and combine several existing research efforts, and identify problems that have not been addressed by the research community before.

We note that Web caching is an area closely related to replication. In caching, whenever a client requests a document for the first time, the client process or the local server handling the request will fetch a copy from the document's server. Before passing it to the client, the document is stored locally in a cache. Whenever that document is requested again, it can be fetched from the cache locally. In replication, a document's server pro-actively places copies of document at various servers, anticipating that enough clients will make use of this copy. Caching and replication thus differ only in the method of creation of copies. Hence, we perceive caching infrastructures (like, for example, Akamai [Dilley et al., 2002]) also as replica hosting systems, as document distribution is initiated by the server. For more information on traditional Web caching, see [Wang, 1999]. A survey on hierarchical and distributed Web caching can be found in [Rodriguez et al., 2001].

A complete design of a Web replica hosting system cannot restrict itself to addressing the above five issues, but should also consider other non-functional

aspects such as *security* and *fault tolerance*. However, in this thesis, we mainly focus on solutions to improve the performance of Web sites. Research addressing security and fault tolerance issues are not presented in this chapter. Moreover, discussing these issues makes sense only in the context of the above five design issues. Therefore, in what follows, we only occasionally refer to these nonfunctional aspects of system design.

The rest of the chapter is organized as follows. In Section 2.2 we present our framework of wide-area replica hosting systems. In Sections 2.3 to 2.7, we discuss each of the above mentioned five problems forming the framework. For each problem, we refer to some of the significant related research efforts, and show how the problem was tackled. Section 2.8 discusses some of the significant research efforts that address the problem of hosting database-driven Web applications. We draw our conclusions in Section 2.9.

2.2. FRAMEWORK

The goal of a replica hosting system is to provide its clients with the best available performance while consuming as little resources as possible. For example, hosting replicas of an object on many servers spread throughout the Internet can decrease the client end-to-end latency, but is bound to increase the operational cost of the system. Replication can also introduce costs and difficulties in maintaining consistency among replicas, but the system should always continue to meet application-specific consistency constraints. The design of a replica hosting system is the result of compromises between performance, cost, and application requirements.

2.2.1. Objective function

In a sense, we are dealing with an optimization problem, which can be modeled by means of an abstract *objective function*, F_{ideal}, whose value λ is dependent on many input parameters:

$$\lambda = F_{ideal}(p_1, p_2, p_3, \ldots, p_n)$$

In our case, the objective function takes two types of input parameters. The first type consists of uncontrollable system parameters, which cannot be directly controlled by the replica hosting system. Typical examples of such uncontrollable parameters are client request rates, update rates for Web documents, and available network bandwidth. The second type of input parameters are those whose value

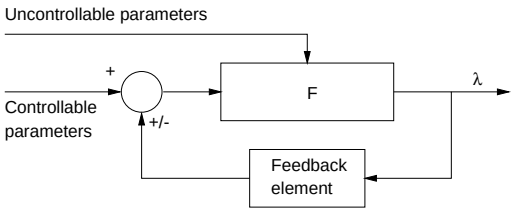


Figure 2.1: The feedback control loop for a replica hosting system.

can be controlled by the system. Examples of such parameters include the number of replicas, the location of replicas, and the adopted consistency protocols.

One of the problems that replica hosting systems are confronted with is to achieve optimal performance with only being able to manipulate the controllable parameters. As a result, continuous feedback is necessary, resulting in a traditional feedback control system as shown in Figure 2.1.

Unfortunately, the actual objective function F_{ideal} represents an ideal situation, in the sense that the function is generally only implicitly known. For example, the actual dimension of λ may be a complex combination of monetary revenues, network performance metrics, and so on. Moreover, the exact relationship between input parameters and the observed value λ may be impossible to derive. Therefore, a different approach is always followed by constructing an objective function F whose output λ is compared to an assumed optimal value λ^* of F_{ideal}. The closer λ is to λ^*, the better. In general, the system is considered to be in an *acceptable state*, if $|\lambda^* - \lambda| \leq \delta$, for some system-dependent value δ.

We perceive any large-scale Web replica hosting system to be constantly adjusting its controllable parameters to keep λ within the acceptable interval around λ^*. For example, during a flash crowd (a sudden and huge increase in the client request rate), a server's load increases, in turn increasing the time needed to service a client. These effects may result in λ falling out of the acceptable interval and that the system must adjust its controllable parameters to bring λ back to an acceptable value. The actions on controllable parameters can be such as increasing the number of replicas, or placing replicas close to the locations that generate most requests. The exact definition of the objective function F, its input parameters, the optimal value λ^*, and the value of δ are defined by the system designers and will generally be based on application requirements and constraints such as cost.

We use the notion of an objective function to describe the different components of a replica hosting system, corresponding to the different parts of the system design. These components cooperate with each other to optimize λ. They operate on the controllable parameters of the objective function, or observe its uncontrollable parameters.

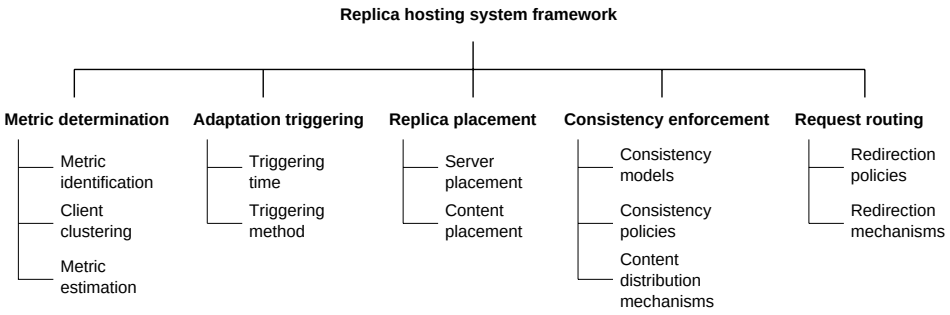


Figure 2.2: A framework for evaluating wide-area replica hosting systems.

2.2.2. Framework elements

We identify five main issues that have to be considered during the design of a replica hosting system: metric determination, adaptation triggering, replica placement, consistency enforcement, and request routing. These issues can be treated as chronologically ordered steps that have to be taken when transforming a centralized service into a replicated one. Our proposed framework of a replica hosting system matches these five issues as depicted in Figure 2.2. Below we discuss the five issues and show how each of them is related to the objective function.

In **metric determination**, we address the question how to find and estimate the metrics required by different components of the system. Metric determination is the problem of estimating the value of the objective function parameters. We discuss two important issues related to metric estimation that need to be addressed to build a good replica hosting system. The first issue is *metric identification*: the process of identifying the metrics that constitute the objective function the system aims to optimize. For example, a system might want to minimize client latency to attract more customers, or might want to minimize the cost of replication. The other important issue is the process of *metric estimation*. This involves the design of mechanisms and services related to estimation or measurement of metrics in a scalable manner. As a concrete example, measuring client latency to *every* client is generally not scalable. In this case, we need to group clients into clusters and measure client-related metrics on a per-cluster basis instead of on a per-client basis (we call this process of grouping clients *client clustering*). In general, the metric estimation component measures various metrics needed by other components of the replica hosting system.

Adaptation triggering addresses the question when to adjust or adapt the system configuration. In other words, we define when and how we can detect that λ has drifted too much from λ^*. Consider a flash crowd causing poor client

latency. The system must identify such a situation and react, for example, by increasing the number of replicas to handle the increase in the number requests. Similarly, congestion in a network where a replica is hosted can result in poor accessibility of that replica. The system must identify such a situation and possibly move that replica to another server. The adaptation-triggering mechanisms do not form an input parameter of the objective function. Instead, they form the heart of the feedback element in Figure 2.1, thus indirectly control λ and maintain the system in an acceptable state.

With **Replica placement** we address the question where to place replicas. This issue mainly concerns two problems: selection of locations to install replica servers that can host replicas (*replica server placement*) and selection of replica servers to host replicas of a given object (*replica content placement*). The server placement problem must be addressed during the initial infrastructure installation and during the hosting infrastructure upgrading. The replica content placement algorithms are executed to ensure that content placement results in an acceptable value of λ, given a set of replica servers. Replica placement components use metric estimation services to get the value of metrics required by their placement algorithms. Both *replica server placement* and *replica content placement* form controllable input parameters of the objective function.

With **consistency enforcement** we consider how to keep the replicas of a given object consistent. Maintaining consistency among replicas adds overhead to the system, particularly when the application requires strong consistency (meaning clients are intolerant to stale data) and the number of replicas is large. The problem of consistency enforcement is defined as follows. Given certain application consistency requirements, we must decide which *consistency models, consistency policies* and *content distribution mechanisms* can meet these requirements. A consistency model dictates the consistency-related properties of the content delivered by the systems to its clients. These models define consistency properties of objects based on time, value, or the order of transactions executed on the object. A consistency model is usually adopted by consistency policies, which define how, when, and which content distribution mechanisms must be applied. The content distribution mechanisms specify the protocols by which replica servers exchange updates. For example, a system can adopt a time-based consistency model and employ a policy where it guarantees its clients that it will never serve a replica that is more than an hour older than the most recent state of the object.

Request routing is about deciding how to direct clients to the replicas they need. We choose from a variety of *redirection policies* and *redirection mechanisms*. Whereas the mechanisms provide a method for informing clients about replica locations, the policies are responsible for determining which replica must serve a client. The request routing problem is complementary to the placement

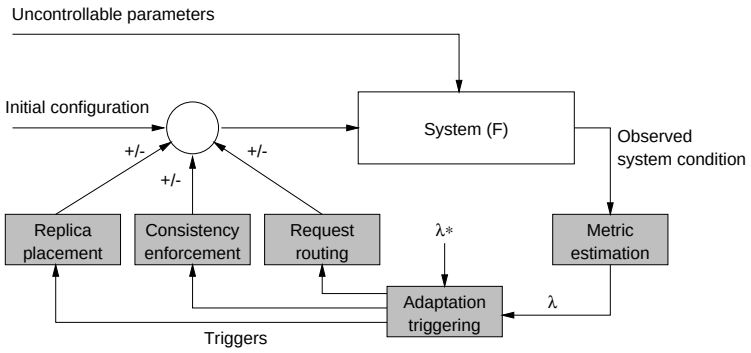


Figure 2.3: Interactions between different components of a wide-area replica hosting system.

problem, as the assumptions made when solving the latter are implemented by the former. For example, we can place replica servers close to our clients, assuming that the redirection policy directs the clients to their nearby replica servers. However, deliberately drifting away from these assumptions can sometimes help in optimizing the objective function. For example, we may decide to direct some clients to more distant replica servers to offload the client-closest one. Therefore, we treat *request routing* as one of the (controllable) objective function parameters.

Each of the above design issues corresponds to a single *logical* system component. How each of them is actually realized can be very different. The five components together should form a scalable Web replica hosting system. The interaction between these components is depicted in Figure 2.3, which is a refinement of our initial feedback control system shown in Figure 2.1. We assume that λ^* is a function of the uncontrollable input parameters, that is:

$$\lambda^* = \min_{p_{k+1},\ldots,p_n} F(\underbrace{p_1,\ldots,p_k}_{\text{Uncontrollable parameters}}, \underbrace{p_{k+1},\ldots,p_n}_{\text{Controllable parameters}})$$

Its value is used for adaptation triggering. If the difference with the computed value λ is too high, the triggering component initiates changes in one or more of the three *control components*: replica placement, consistency enforcement, or request routing. These different components strive to maintain λ close to λ^*. They manage the controllable parameters of the objective function, now represented by the actually built system. Of course, the system conditions are also influenced by the uncontrollable parameters. The system condition is measured by the metric estimation services. They produce the current system value λ, which is then passed to the adaptation triggering component for subsequent comparison. This process of adaptation continues throughout the system's lifetime.

Note that the metric estimation services are also being used by components for replica placement, consistency enforcement, and request routing, respectively, for deciding on the quality of their decisions. These interactions are not shown in the figure for sake of clarity.

2.3. METRIC DETERMINATION

The metric determination component is required to measure the system condition. It allows the system to detect when the system quality drifts away from the acceptable interval so that the system can adapt its configuration if necessary. Another purpose of the metric determination component is to provide each of the three control components with measurements of their input data. For example, replica placement algorithms may need latency measurements in order to generate a placement that is likely to minimize the average latency suffered by the clients. Similarly, consistency enforcement algorithms might require information on object staleness in order to react with switching to stricter consistency mechanisms. Finally, request routing policies may need to know the current load of replica servers in order to distribute the requests currently targeting heavily loaded servers to less loaded ones.

In this section, we discuss three issues that have to be addressed to enable scalable metric determination. The first issue is *metric selection*. Depending on the performance optimization criteria, a number of metrics must be carefully selected to accurately reflect the behavior of the system. Section 2.3.1 discusses metrics related to client latency, network distance, network usage, object hosting cost, and consistency enforcement.

The second issue is *client clustering*. Some client-related metrics should ideally be measured separately for each client. However, this can lead to scalability problems as the number of clients for a typical wide-area replica hosting system can be in the order of millions. A common solution to address this problem is to group clients into clusters and measure client-related metrics on a per-cluster basis. Section 2.3.2 discusses various client clustering schemes.

The third issue is *metric estimation* itself. We must choose mechanisms to collect metric data. These mechanisms typically use client-clustering schemes to estimate client-related metrics. Section 2.3.3 discusses some popular mechanisms that collect metric data.

2.3.1. Choice of metrics

The choice of metrics must reflect all aspects of the desired performance. First of all, the system must evaluate all metrics that take part in the computation of

Class	Description
Temporal	The metric reflects how long a certain action takes
Spatial	The metric is expressed in terms of a distance that is related to the topology of the underlying network, or region in which the network lies
Usage	The metric is expressed in terms of usage of resources of the underlying network, notably consumed bandwidth
Financial	Financial metrics are expressed in terms of a monetary unit, reflecting the monetary costs of deploying or using services of the replica hosting system
Consistency	The metrics express to what extent a replica's value may differ from the master copy

Table 2.1: Five different classes of metrics used to evaluate performance in replica hosting systems.

the objective function. Additionally, the system also needs to measure some extra metrics needed by the control components. For example, a map of host-to-host distances may help the replica placement algorithms, although it does not have to be used by the objective function.

There exists a wide range of metrics that can reflect the requirements of both the system's clients and the system's operator. For example, the metrics related to latency, distance, and consistency can help evaluate the client-perceived performance. Similarly, the metrics related to network usage and object hosting cost are required to control the overall system maintenance cost, which should remain within bounds defined by the system's operator. We distinguish five classes of metrics, as shown in Figure 2.1, and which are discussed in the following sections.

Temporal metrics

An important class of metrics is related to the *time* it takes for peers to communicate, generally referred to as latency metrics. Latency can be defined in different ways. To explain, we consider a client-server system and follow the approach described in [Dykes et al., 2000] by modeling the total time to process a request, as seen from the client's perspective, as

$$T = T_{DNS} + T_{conn} + T_{res} + T_{rest}$$

T_{DNS} is the DNS lookup time needed to find the server's network address. As reported by [Cohen and Kaplan, 2001], DNS lookup time can vary tremendously due to cache misses (i.e., the client's local DNS server does not have the address of the requested host in its cache), although in many cases it stays below 500 milliseconds.

T_{conn} is the time needed to establish a TCP connection, which, depending on the type of protocols used in a replica hosting system, may be relevant to take into account. [Zari et al., 2001] report that T_{conn} will often be below 200 milliseconds, but that, like in the DNS case, very high values up to even 10 seconds may also occur.

T_{res} is the time needed to transfer a request from the client to the server and receiving the first byte of the response. This metric is comparable to measuring the round-trip time (RTT) between two nodes, but includes the time the server needs to handle the incoming request. Finally, T_{rest} is the time needed to complete the transfer of the entire response.

When considering latency, two different versions are often considered. The *end-to-end latency* is taken as the time needed to send a request to the server, and is often taken as $0.5T_{res}$, possibly including the time T_{conn} to setup a connection. The *client-perceived latency* is defined as $T - T_{rest}$. This latter latency metric reflects the real delay observed by a user.

Obtaining accurate values for latency metrics is not a trivial task as it may require specialized mechanisms, or even a complete infrastructure. One particular problem is predicting client-perceived latency, which not only involves measuring the round-trip delay between two nodes (which is independent of the size of the response), but may also require measuring bandwidth to determine T_{rest}. The latter has shown to be particularly cumbersome requiring sophisticated techniques [Lai and Baker, 1999]. We discuss the problem of latency measurement further in Section 2.3.3.

Spatial metrics

As an alternative to temporal metrics, many systems consider a spatial metric such as number of network-level hops or hops between autonomous systems, or even the geographical distance between two nodes. In these cases, the underlying assumption is generally that there exists a map of the network in which the spatial metric can be expressed.

Maps can have different levels of accuracy. Some maps depict the Internet as a graph of Autonomous Systems (ASes), thus unifying all machines belonging to the same AS. They are used for example by [Pierre et al., 2002]. The graph of ASes is relatively simple and easy to operate on. However, because ASes significantly vary in size, this approach can suffer from inaccuracy. Other maps depict the Internet as a graph of routers, thus unifying all machines connected to the same router [Pansiot and Grad, 1998]. These maps are more detailed than the AS-based ones, but are not satisfying predictors for latency. For example, [Huffaker et al., 2002] found that the number of router hops is accurate in selecting the closest server in only 60% of the cases. The accuracy of router-level maps can

be expected to decrease in the future with the adoption of new routing technologies such as Multi-Protocol Label Switching (MPLS) [Rosen et al., 2001], which may hide the routing paths within a network. Finally, some systems use proprietary distance calculation schemes, for example by combining the two above approaches [Rabinovich and Aggarwal, 1999].

[Huffaker et al., 2002] examined to what extent geographical distance could be used instead of latency metrics. They showed that there is generally a close correlation between geographical distance and RTT. An earlier study using simple network measurement tools by [Ballintijn et al., 2000], however, reported only a weak correlation between geographical distance and RTT. This difference may be caused by the fact that many more monitoring points *outside* the U.S. were used, but that many physical connections actually cross through networks located *in* the U.S. This phenomenon also caused large deviations in the results by [Huffaker et al., 2002]

An interesting approach based on geographical distance is followed in the Global Network Positioning (GNP) project [Ng and Zhang, 2002]. In this case, the Internet is modeled as an N-dimensional geometric space. GNP is used to estimate the latency between two arbitrary nodes. We describe GNP and its several variants in more detail in Section 2.3.3 when discussing metric estimation services.

Constructing and exploiting a map of the Internet is easier than running an infrastructure for latency measurements. The maps can be derived, for example, from routing tables. Interestingly, [Crovella and Carter, 1995] reported that the correlation between the distance in terms of hops and the latency is quite poor. However, other studies show that the situation has changed. [McManus, 1999] shows that the number of hops between ASes can be used as an indicator for client-perceived latency. Research reported in [Obraczka and Silva, 2000] revealed that the correlation between the number of network-level or AS-level hops and round-trip times (RTTs) has further increased, but that RTT still remains the best choice when a single latency metric is needed for measuring client-perceived performance.

Network usage metrics

Another important metric is the total amount of consumed network resources. Such resources could include routers and other network elements, but often entails only the consumed bandwidth. The total network usage can be classified into two types. Internal usage is caused by the communication between replica servers to keep replicas consistent. External usage is caused by communication between clients and replica servers. Preferably, the ratio between external and internal usage is high, as internal usage can be viewed as a form of overhead introduced merely to keep replicas consistent. On the other hand, overall network usage may

decrease in comparison to the non-replicated case, but may require measuring more than, for example, consumed bandwidth only.

To see the problem at hand, consider a non-replicated document of size s bytes that is requested r times per seconds. The total consumed bandwidth in this case is $r \cdot s$, plus the cost of r separate connections to the server. The cost of a connection can typically be expressed as a combination of setup costs and the average distance that each packet associated with that connection needs to travel. Assume that the distance is measured in the number of hops (which is reasonable when considering network usage). In that case, if l_r is the average length of a connection, we can also express the total consumed bandwidth for reading the document as $r \cdot s \cdot l_r$.

On the other hand, suppose the document is updated w times per second and that updates are always immediately pushed to, say, n replicas. If the average path length for a connection from the server to a replica is l_w, update costs are $w \cdot s \cdot l_w$. However, the average path length of a connection for reading a document will now be lower in comparison to the non-replicated case. If we assume that $l_r = l_w$, the total network usage may change by a factor w/r in comparison to the non-replicated case.

Of course, more precise models should be applied in this case, but the example illustrates that merely measuring consumed bandwidth may not be enough to properly determine network usage. This aspect becomes even more important given that pricing may be an issue for providing hosting services, an aspect that we discuss next.

Financial metrics

Of a completely different nature are metrics that deal with the economics of content delivery networks. To date, such metrics form a relatively unexplored area, although there is clearly interest to increase our insight (see, for example, [Janiga et al., 2001]). We need to distinguish at least two different roles. First, the owner of the hosting service is confronted with costs for developing and maintaining the hosting service. In particular, costs will be concerned with server placement, server capacity, and network resources (see, e.g., [Chandra et al., 2001]). This calls for metrics aimed at the hosting service provider.

The second role is that of customers of the hosting service. Considering that we are dealing with shared resources that are managed by a service provider, accounting management by which a precise record of resource consumption is developed, is important for billing customers [Aboba et al., 2000]. However, developing pricing models is not trivial and it may turn out that simple pricing schemes will dominate the sophisticated ones, even if application of the latter are cheaper for customers [Odlyzko, 2001]. For example, Akamai uses peak consumed bandwidth as its pricing metric.

The pricing model for hosting an object can directly affect the control components. For example, a model can mandate that the number of replicas of an object is constrained by the money paid by the object owner. Likewise, there exist various models that help in determining object hosting costs. Examples include a model with a flat base fee and a price linearly increasing along with the number of object replicas, and a model charging for the total number of clients serviced by all the object replicas.

We observe that neither financial metrics for the hosting service provider nor those for consumers have actually been established other than in some ad hoc and specific fashion. We believe that developing such metrics, and notably the models to support them, is one of the more challenging and interesting areas for content delivery networks.

Consistency metrics

Consistency metrics inform to what extent the replicas retrieved by the clients are consistent with the replica version that was up-to-date at the moment of retrieval. Many consistency metrics have been proposed and are currently in use, but they are usually quantified along three different axes.

In *time-based* consistency models, the difference between two replicas A and B is measured as the time between the latest update on A and the one on B. In effect, time-based models measure the staleness of a replica in comparison to another, more recently updated replica. Taking time as a consistency metric is popular in Web-hosting systems as it is easy to implement and independent of the semantics of the replicated object. Because updates are generally performed at only a single primary copy from where they are propagated to secondaries, it is easy to associate a single timestamp with each update and to subsequently measure the staleness of a replica.

In *value-based* models, it is assumed that each replica has an associated numerical value that represents its current content. Consistency is then measured as the numerical difference between two replicas. This metric requires that the semantics of the replicated object are known or otherwise it would be impossible to associate and compare object values. An example of where value-based metrics can be applied is a stock-market Web document containing the current values of shares. In such a case, we could define a Web document to be inconsistent if at least one of the displayed shares differs by more than 2% with the most recent value.

Finally, in *order-based* models, reads and writes are perceived as transactions and replicas can only differ in the order of execution of write transactions according to certain constraints. These constraints can be defined as the allowed number of out-of-order transactions, but can also be based on the dependencies

between transactions as is commonly the case for distributed shared-memory systems [Mosberger, 1993], or client-centric consistency models as introduced in Bayou [Terry et al., 1994].

Metric classification

Metrics can be classified into two types: static and dynamic. Static metrics are those whose estimates do not vary with time, as opposed to dynamic metrics. Metrics such as the geographical distance are static in nature, whereas metrics such as end-to-end latency, number of router hops or network usage are dynamic. The estimation of dynamic metrics can be a difficult problem as it must be performed regularly to be accurate. Note that determining how often a metric should be estimated is a problem by itself. For example, [Paxson, 1997a] found that the time periods over which end-to-end routes persist vary from seconds to days.

Dynamic metrics can be more useful when selecting a replica for a given client as they estimate the current situation. For example, [Crovella and Carter, 1995] conclude that the use of a dynamic metric instead of a static one is more useful for replica selection as the former can also account for dynamic factors such as network congestion. Static metrics, in turn, are likely to be exploited by replica server placement algorithms as they tend to be more directed toward a global, long-lasting situation than an instantaneous one.

In general, however, any combination of metrics can be used by any control component. For example, the placement algorithms proposed by [Radoslavov et al., 2001] and [Qiu et al., 2001] use dynamic metrics (end-to-end latency and network usage). Also [Dilley et al., 2002] and [Rabinovich and Aggarwal, 1999] use end-to-end latency as a primary metric for determining the replica location. Finally, the request-routing algorithm described in [Szymaniak et al., 2003] exploits network distance measurements. We observe that the existing systems tend to support a small set of metrics, and use all of them in each control component.

2.3.2. Client clustering

As we noticed before, some metrics should be ideally measured on a per-client basis. In a wide-area replica hosting system, for which we can expect millions of clients, this poses a scalability problem to the estimation services as well as the components that need to use them. Hence, there is a need for scalable mechanisms for metric estimation.

A popular approach by which scalability is achieved is *client clustering* in which clients are grouped into clusters. Metrics are then estimated on a per-cluster basis instead of on a per-client basis. Although this solution allows to estimate metrics in a scalable manner, the efficiency of the estimation depends on

the accuracy of clustering mechanisms. The underlying assumption here is that the metric value computed for a cluster is representative of values that would be computed for each individual client in that cluster. Accurate clustering schemes are those which keep this assumption valid.

The choice of a clustering scheme depends on the metric it aims to estimate. Below, we present different kinds of clustering schemes that have been proposed in the literature.

Local name servers

Each Internet client contacts its local DNS server to resolve a service host name to its IP address(es). The clustering scheme based on local name servers unifies clients contacting the same name server, as they are assumed to be located in the same network-topological region. This is a useful abstraction as DNS-based request-routing schemes are already used in the Internet. However, the success of these schemes relies on the assumption that clients and local name servers are close to each other. [Shaikh et al., 2001] performed a study on the proximity of clients and their name servers based on the HTTP logs from several commercial Web sites. Their study concludes that clients are typically eight or more hops from their representative name servers. The authors also measured the round trip times both from the name servers to the servers (name-server latency) and from the clients to the servers (client latency). It turns out that the correlation between the name-server latency and the actual client latency is quite poor. They conclude that the latency measurements to the name servers are only a weak approximation of the latency to actual clients. These findings have been confirmed by [Mao et al., 2002].

Autonomous Systems

The Internet has been built as a graph of individual network domains, called Autonomous Systems (ASes). The AS clustering scheme groups together clients located in the same AS, as is done for example, by [Pierre et al., 2002]. This scheme naturally matches the AS-based distance metric. Further clustering can be achieved by grouping ASes into a hierarchy, as proposed by [Barford et al., 2001], which in turn can be used to place caches.

Although an AS is usually formed out of a set of networks belonging to a single administrative domain, it does not necessarily mean that these networks are proximal to each other. Therefore, estimating latencies with an AS-based clustering scheme can lead to poor results. Furthermore, since ASes are global in scope, multiple ASes may cover the same geographical area. It is often the case that some IP hosts are very close to each other (either in terms of latency or hops)

but belong to different ASes, while other IP hosts are very far apart but belong to the same AS. This makes the AS-based clustering schemes not very effective for proximity-based metric estimations.

Client proxies

In some cases, clients connect to the Internet through *proxies* which provide them with services such as Web caching and prefetching. Client proxy-based clustering schemes group together all clients using the same proxy into a single cluster. Proxy-based schemes can be useful to measure latency if the clients are close to their proxy servers. An important problem with this scheme is that many clients in the Internet do not use proxies at all. Thus, this clustering scheme will create many clusters consisting of only a single client, which is inefficient with respect to achieving scalability for metric estimation.

Network-aware clustering

Researchers have proposed another scheme for clustering Web clients, which is based on client-network characteristics. [Krishnamurthy and Wang, 2000] evaluate the effectiveness of a simple mechanism that groups clients having the same first three bytes of their IP addresses into a single cluster. However, this simple mechanism fails in more than 50% of the cases when checking whether grouped clients actually belong to the same network. The authors identify two reasons for failure. First, their scheme wrongly merges small clusters that share the same first three bytes of IP addresses as a single class-C network. Second, it splits several class-A, class-B, and CIDR networks into multiple class-C networks. Therefore, the authors propose a novel method to identify clusters by using the prefixes and network masks information extracted from the Border Gateway Protocol routing tables [Rekhter and Li, 1995]. The proposed mechanism consists of the following steps:

1. Creating a merged prefix table from routing table snapshots

2. Performing the longest prefix matching on each client IP address (as routers do) using the constructed prefix table

3. Classifying all the clients which have the same longest prefix into a single cluster.

The authors demonstrate the effectiveness of their approach by showing a success rate of 99.99% in their validation tests.

Hierarchical clustering

Most clustering schemes aim at achieving a scalable manner of metric estimation. However, if the clusters are too coarse grained, it decreases the accuracy of measurement simply because the underlying assumption that the difference between the metric estimated to a cluster and to a client is negligible is no longer valid. Hierarchical clustering schemes help in estimating metrics at different levels (such as intra-cluster and inter-cluster), thereby aiming at improving the accuracy of measurement, as in IDMaps [Francis et al., 2001] and Radar [Rabinovich and Aggarwal, 1999]. Performing metric estimations at different levels results not only in better accuracy, but also in better scalability.

Note that there can be other possible schemes of client clustering, based not only on the clients' network addresses or their geographical proximities, but also on their content interests (see, e.g., [Xiao and Zhang, 2001]). However, such clustering is not primarily related to improving scalability through replication, for which reason we further exclude it from our study.

2.3.3. Metric estimation services

Once the clients are grouped into their respective clusters, the next step is to obtain the values for metrics (such as latency or network overhead). Estimation of metrics on a wide-area scale such as the Internet is not a trivial task and has been addressed by several research initiatives before [Francis et al., 2001; Moore et al., 1996]. In this section, we discuss the challenges involved in obtaining the value for these metrics.

Metric estimation services are responsible for providing values for the various metrics required by the system. These services aid the control components in taking their decisions. For example, these services can provide replica placement algorithms with a map of the Internet. Also, metric estimation services can use client-clustering schemes to achieve scalability.

Metric estimations schemes can be divided into two groups: *active* and *passive* schemes. Active schemes obtain respective metric data by simulating clients and measuring the performance observed by these simulated clients. Active schemes are usually highly accurate, but these simulations introduce additional load to the replica hosting system. Examples of active mechanisms are Cprobes [Carter and Crovella, 1997] and Packet Bunch Mode [Paxson, 1997b]. Passive mechanisms obtain the metric data from observations of existing system behavior. Passive schemes do not introduce additional load to the network, but deriving the metric data from the past events can suffer from poor accuracy. Examples of passive mechanisms include SPAND [Stemm et al., 2000] and EtE [Fu et al., 2002].

Different metrics are by nature estimated in different manners. For example,

metric estimation services are commonly used to measure client latency or network distance. The consistency-related metrics are not measured by a separate metric estimation service, but are usually measured by instrumenting client applications. In this section, our discussion of existing research efforts mainly covers services that estimate network-related metrics.

IDMaps

IDMaps is an active service that aims at providing an architecture for measuring and disseminating distance information across the Internet [Francis et al., 1999, 2001]. IDMaps uses programs called tracers that collect and advertise distance information as so-called distance maps. IDMaps builds its own client-clustering scheme. It groups different geographical regions as boxes and constructs distance maps between these boxes. The number of boxes in the Internet is relatively small (in the order of thousands). Therefore, building a distance table between these boxes is inexpensive. To measure client-server distance, an IDMaps client must calculate the distance to its own box and the distance from the target server to this server's box. Given these two calculations, and the distance between the boxes calculated based on distance maps, the client can discover its real distance to the server. It must be noted that the efficiency of IDMaps heavily depends on the size and placement of boxes.

King

King is an active metric estimation method [Gummadi et al., 2002]. It exploits the global infrastructure of DNS servers to measure the latency between two arbitrary hosts. King approximates the latency between two hosts, H_1 and H_2, with the latency between their local DNS servers, S_1 and S_2.

Assume that a host X needs to calculate the latency between hosts H_1 and H_2. The latency between their local DNS servers, $L_{S_1S_2}$, is calculated based on round-trip times (RTTs) of two DNS queries. With the first query, host X queries the DNS server S_1 about some non-existing DNS name that belongs to a domain hosted by S_1 [see Figure 2.4(a)]. In this way, X discovers its latency to S_1:

$$L_{XS_1} = \frac{1}{2} RTT_1$$

By querying about a non-existing name, X ensures that the response is retrieved from S_1, as no cached copy of that response can be found anywhere in the DNS.

With the second query, host X queries the DNS server S_1 about another non-existing DNS name that this time belongs to a domain hosted by S_2 (see Figure 2.4b). In this way, X measures the latency of its route to S_2 that goes through

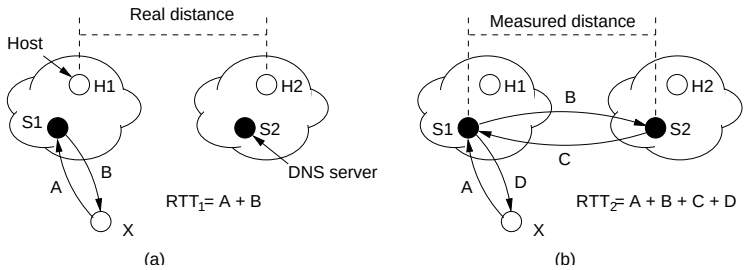


Figure 2.4: The two DNS queries of King

S_1:

$$L_{XS_2} = \frac{1}{2}RTT_2$$

A crucial observation is that this latency is a sum of two partial latencies, one between X and S_1, and the other between S_1 and S_2: $L_{XS_2} = L_{XS_1} + L_{S_1S_2}$. Since L_{XS_1} has been measured by the first DNS query, X may subtract it from the total latency L_{XS_2} to determine the latency between the DNS servers:

$$L_{S_1S_2} = L_{XS_2} - L_{XS_1} = \frac{1}{2}RTT_2 - \frac{1}{2}RTT_1$$

Note that S_1 will forward the second query to S_2 only if S_1 is configured to accept so-called "recursive" queries from X [Mockapetris, 1987b].

In essence, King is actively probing with DNS queries. A potential problem with this approach is that an extensive use of King may result in overloading the global infrastructure of DNS servers. In such case, the efficiency of DNS is likely to decrease, which can degrade the performance of the entire Internet. Also, according to the DNS specification, it is recommended to reject recursive DNS queries that come from non-local clients, which renders many DNS servers unusable for King [Mockapetris, 1987a].

Network Positioning

The idea of network positioning has been proposed in [Ng and Zhang, 2002], where it is called Global Network Positioning (GNP). GNP is a novel approach to the problem of network distance estimation, where the Internet is modeled as an N-dimensional geometric space. GNP approximates the latency between any two hosts as the Euclidean distance between their corresponding positions in the geometric space.

GNP relies on the assumption that latencies can be triangulated in the Internet. The position of any host X is computed based on its measured latencies between X and k *landmark* hosts, whose positions have been computed earlier ($k \geq N+1$,

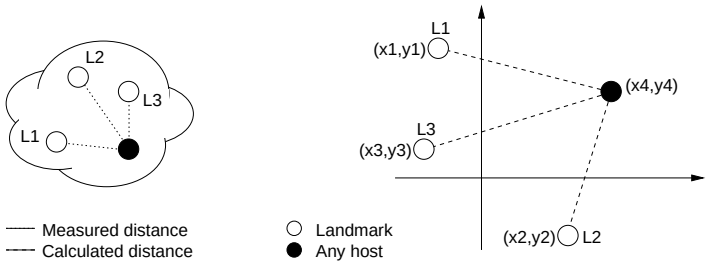


Figure 2.5: Positioning in GNP

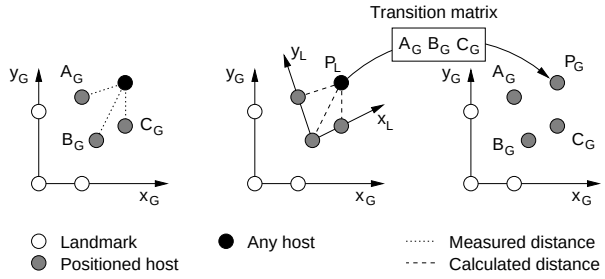


Figure 2.6: Positioning in Lighthouses

to ensure that the calculated position is unique). By treating these latencies as distances, GNP triangulates the position of X (see Figure 2.5). The triangulation is implemented by means of Simplex-downhill, which is a classical optimization method for multi-dimensional functions [Nelder and Mead, 1965].

The most important limitation of GNP is that the set of landmarks can never change. If any of them becomes unavailable, the latency to that landmark cannot be measured and GNP is no longer able to position any more hosts. It makes GNP sensitive to landmark failures.

This limitation is removed in the Lighthouses system [Pias et al., 2003]. The authors have shown that hosts can be accurately positioned relative to any previously positioned hosts, acting as "local" landmarks. This eliminates the need for contacting the original landmarks each time a host is positioned (see Figure 2.6). It also allows to improve the positioning accuracy, by selecting some of the local landmarks close to the positioned host [Castro et al., 2003].

SCoLE further improves the scalability of the system by allowing each host to select its own positioning parameters, construct its own "private" space, and position other hosts in that space [Szymaniak et al., 2004]. This effectively removes the necessity of a global negotiation to determine positioning parameters, such as the space dimension, the selection of global landmarks, and the positioning algorithm. Such an agreement is difficult to reach in large-scale systems, where

different hosts can have different requirements with respect to the latency estimation process. Latency estimates performed in different private spaces have been shown to be highly correlated, even though these spaces have completely different parameters.

Another approach is to position all hosts simultaneously as a result of a global optimization process [Cox et al., 2003; Shavitt and Tankel, 2003; Waldvogel and Rinaldi, 2003]. In this case, there is no need to choose landmarks, since every host is in fact considered to be a landmark. The global optimization approach is generally faster than its iterative counterpart, which positions hosts one by one. The authors also claim that it leads to better accuracy, and that it is easy to implement in a completely distributed fashion. However, because it operates on all the latencies simultaneously, it can potentially have to be re-run every time new latency measurements are available. Such a re-run is likely to be computationally expensive in large-scale systems, where the number of performed latency measurements is high. Note that the network position approaches can be applied only where the modeled metric can be triangulated, making it more difficult when measuring, for example, bandwidth.

SPAND

SPAND is a shared passive network performance measurement service [Stemm et al., 2000]. This service aims at providing network-related measures such as client end-to-end latency, available bandwidth, or even application-specific performance details such as access time for a Web object. The components of SPAND are client applications that can log their performance details, a packet-capturing host that logs performance details for SPAND-unaware clients, and performance servers that process the logs sent by the above two components. The performance servers can reply to queries concerning various network-related and application-specific metrics. SPAND has an advantage of being able to produce accurate application-specific metrics if there are several clients using that application in the same shared network. Further, since it employs passive measurement, it does not introduce any additional traffic.

Network Weather Service

The Network Weather Service (NWS) is an active measurement service [Wolski et al., 1999]. It is primarily used for Grid computing, where decisions regarding scheduling of distributed computations are made based on the knowledge of server loads and several network performance metrics, such as available bandwidth and end-to-end latency. Apart from measuring these metrics, it also employs prediction mechanisms to forecast their value based on past events. In NWS, the metrics

are measured using special sensor processes, deployed on every potential server node. Further, to measure end-to-end latency active probes are sent between these sensors. NWS uses adaptive forecasting approach, in which the service dynamically identifies the model that gives the least prediction error. NWS has also been used for replica selection [McCune and Andresen, 1998]. However, exploiting NWS directly by a wide-area replica hosting system can be difficult, as this service does not scale to the level of the Internet. This is due to the fact that it runs sensors in every node and does not use any explicit client clustering schemes. On the other hand, when combined with a good client clustering scheme and careful sensor placement, NWS can become a useful metric estimation service.

Akamai metric estimation

Commercial replica hosting systems often use their own monitoring or metric estimation services. Akamai has built its own distributed monitoring service to monitor server resources, network-related metrics and overall system performance. The monitoring system places monitors in every replica server to measure server resources. The monitoring system simulates clients to determine if the overall system performance is in an acceptable state as perceived by clients. It measures network-related information by employing agents that communicate with border routers in the Internet as peers and derive the distance-related metrics to be used for its placement decisions [Dilley et al., 2002].

Other systems

In addition to the above wide-area metric estimation systems, there are different classes of systems that measure service-related metrics such as content popularity, client-aborted transfers, and amount of consumed bandwidth. These kinds of systems perform estimation in a smaller scale, and mostly measure metrics as seen by a single server.

Web page instrumentation and associated code (e.g., in Javascript) is being used in various commercial tools for measuring service-related metrics. In these schemes, instrumentation code is downloaded by the client browser after which it tracks the download time for individual objects and reports performance characteristics to the Web site.

EtE is a passive system used for measuring metrics such as access latency, and content popularity for the contents hosted by a server [Fu et al., 2002]. This is done by running a special model near the analyzed server that monitors all the service-related traffic. It is capable of determining sources of delay (distinguishing between network and server delays), content popularity, client-aborted transfers and the impact of remote caching on the server performance.

[Rabinovich et al., 2006] propose a marketplace-driven measurement called DipZoom. The system aims to provide focused, on-demand Internet measurements for large scale distributed systems. Compared to previous approaches that require a well-established infrastructure for measurement, DipZoom offers a match-making service that allows different measurement providers to trade their services. It uses peer-to-peer concepts to bring together experimenters in need of measurements with external measurement providers. It then harnesses market forces to orchestrate the supply and demand to provide a "free market" eco-system. This approach can be potentially useful for applications that do not have the luxury of having a well established infrastructure for running metric estimation services.

2.3.4. Discussion

In this section, we discussed three issues related to metric estimation: metric selection, client clustering, and metric estimation.

Metric selection deals with deciding which metrics are important to evaluate system performance. In most cases, optimizing latency is considered to be most important in many works. Another metric used to measure the efficiency of a system is consumed bandwidth. However, in order to measure the efficiency of a consistency protocol as expressed by the ratio between the consumed bandwidth for replica updates and the bandwidth delivered to clients, some distance metric needs to be taken into account as well. When it comes to consistency metrics, three different types need to be considered: those related to time, value, and the ordering of operations. It appears that this differentiation is fairly complete, leaving the actual implementation of consistency models and the enforcement of consistency the main problem to solve.

A scalability problem that these metrics introduce is that they, in theory, require measurements on a per-client basis. With millions of potential clients, such measurements are impossible. This problem is alleviated by client-clustering schemes. Finding the right metric for clustering, and also one that can be easily established has shown to be difficult. However, network-aware clustering by which a prefix of the network address is taken as criterion for clustering has led to very accurate results.

Once a metric has been chosen, its value needs to be determined. This is where metric estimation services come into place. Various services exist, including some very recent ones that can handle difficult problems such as estimating the latency between two arbitrary nodes in the Internet. We note that the techniques and results presented in this section apply to CDNs hosting both static Web content and dynamic Web applications. Hence, these results are complementary to the solutions addressed by the thesis.

2.4. ADAPTATION TRIGGERING

The performance of a replica hosting system changes with the variations of uncontrollable system parameters such as client access patterns and network conditions. These changes make the current value of the system λ drift away from the optimal value λ^*, and fall out of the acceptable interval. The system needs to maintain a desired level of performance by keeping λ in an acceptable range amidst these changes. The adaptation triggering component of the system is responsible for identifying changes in the system and for adapting the system configuration to bound λ within the acceptable range. This adaptation consists of a combination of changes in replica placement, consistency policy, and request routing policy.

We classify adaptation triggering components along two criteria. First, they can be classified based on their timing nature. Second, they can be classified based on which element of the system actually performs the triggering.

2.4.1. Time-based classification

Taking timing into account, we distinguish three different triggering mechanisms: periodic triggers, aperiodic triggers, and triggers that combine these two.

Periodic triggers

A periodic triggering component analyzes a number of input variables, or λ itself, at fixed time intervals. If the analysis reveals that λ is too far from λ^*, the system triggers the adaptation. Otherwise, it allows the system to continue with the same configuration. Such a periodic evaluation scheme can be effective for systems that have relatively stable uncontrollable parameters. However, if the uncontrollable parameters fluctuate a lot, then it may become very hard to determine a good evaluation periodicity. A too short period will lead to considerable adaptation overhead, whereas a too long period will result in slow reactions to important changes.

Aperiodic triggers

Aperiodic triggers can trigger adaptation at any time. A trigger is usually due to an event indicating a possible drift of λ from the acceptable interval. Such events are often defined as changes of the uncontrollable parameters, such as the client request rates or end-to-end latency, which may reflect issues that the system has to deal with.

The primary advantage of aperiodic triggers is their responsiveness to emergency situations such as flash crowds where the system must be adapted quickly. However, it requires continuous monitoring of metrics that can indicate events in the system, such as server load or client request rate.

Hybrid triggers

Periodic and aperiodic triggers have opposite qualities and drawbacks. Periodic triggers are well suited for detecting slow changes in the system that aperiodic triggers may not detect, whereas aperiodic triggers are well suited to detect emergency situations where immediate action is required. Consequently, a good approach may be a combination of periodic and aperiodic triggering schemes. For example, Radar and Globule use both periodic and aperiodic triggers, which give them the ability to perform global optimizations and to react to emergency situations.

In Radar [Rabinovich and Aggarwal, 1999], every replica server periodically checks for its load and the number of client accesses to a particular replica. An object is deleted for low client accesses and a migration/replication component is invoked if the server load is above a threshold. In addition, a replica server can detect that it is overloaded and ask its replication-managing component to offload it. Adaptation in this case consists either of distributing the load over other replica servers, or to propagate the request to another replication-managing component in case there are not enough replica servers available.

In Globule [Pierre and van Steen, 2006], each primary server periodically evaluates recent client access logs. The need for adapting the replication and consistency policies is determined by this evaluation. Similarly to Radar, each replica server also monitors its request rate and response times. When a server is overloaded, it can request its primary server to reevaluate the replication strategy.

2.4.2. Source-based classification

Adaptation triggering mechanisms also vary upon which part of the system actually performs the triggering. We describe three different kinds of mechanisms.

Server-triggered adaptation

Server-triggered adaptation schemes consider that replica servers are in a good position to evaluate metrics from the system. Therefore, the decision that adaptation is required is taken by one or more replica servers. Radar and Globule use server-triggered adaptation, as they make replica servers monitor and possibly react to system conditions.

Server-triggered adaptation is also well suited for reacting to internal server conditions, such as overloads resulting from flash crowds or denial-of-service (DoS) attacks. For example, in [Jung et al., 2002a], the authors studied the characteristics of flash crowds and DoS attacks. They propose an adaptation scheme where servers can differentiate these two kinds of events and react accordingly: increase the number of replicas in case of a flash crowd, or invoke security mechanisms in case of a DoS attack.

Server-triggered adaptation is effective as the servers are in a good position to determine the need for changes in their strategies in view of other constraints, such as total system cost. Also, these mechanisms do not require running triggering components on elements (hosts, routers) that may not be under the control of the replica hosting system.

Client-triggered adaptation

Adaptation can be triggered by the clients. In client-triggered schemes, clients or client representatives can notice that they experience poor quality of service and request the system to take the appropriate measures. Sayal et al. [Sayal et al., 2003] describe such a system where smart clients provide the servers with feedback information to help take replication decisions. Similarly, Akamai uses emulated clients placed around the world to check whether the response time provided to end users meet the desired level of performance [Dilley et al., 2002].

Client-triggered adaptation can be efficient in terms of preserving a client's QoS. However, it has three important drawbacks. First, the clients or client representatives must cooperate with the replica hosting system. Second, client transparency is lost, as clients or their representatives need to monitor events and take explicit action. Third, by relying on individual clients to trigger adaptation, this scheme may suffer from poor scalability in a wide-area network, unless efficient client clustering methods are used.

Router-triggered adaptation

In router-triggered schemes, adaptation is initiated by the routers that can inform the system of network congestion, link and network failures, or degraded end-to-end request latencies. These schemes observe network-related metrics and operate on them.

Such an adaptation scheme is used in SPREAD [Rodriguez and Sibal, 2000]. In SPREAD, every network has one special router with a distinguished proxy attached to it. If the router notices a TCP communication from a client to retrieve data from a primary server, it intercepts this communication and redirects the request to the proxy attached to it. The proxy gets a copy of the referenced object

from the primary server and services this client and all future requests passing through its network. By using the network layer to implement replication, this scheme builds an architecture that is transparent to the client.

Router-triggered schemes have the advantage that routers are in a good position to observe network-related metrics, such as end-to-end latency and consumed bandwidth while preserving client transparency. Such schemes are useful to detect network congestion or dead links, and thus may trigger changes in replica location. However, they suffer from two disadvantages. First, they require the support of routers, which may not be available to every enterprise building a replica hosting system in the Internet. Second, they introduce an overhead to the network infrastructure, as they need to isolate the traffic targeting Web hosting systems, which involves processing all packets received by the routers.

2.4.3. Discussion

Deciding when to trigger system adaptation is difficult because explicitly computing λ and λ^* may be expensive, if not impossible. This calls for schemes that are both responsive enough to detect the drift of λ from the acceptable interval and computationally inexpensive. This is usually realized by monitoring simple metrics which are believed to significantly influence λ.

Another difficulty is posed by the fact that it is not obvious which adaptive components should be triggered. Depending on the origin of the performance drift, the optimal adaptation may be any combination of changes in replica placement, request routing or consistency policies.

Note that the problem of adaptation for a CDN hosting multi-tiered Web applications is more difficult compared to adapting a single-tiered Web site (which is the case for static Web content). Adapting Web application configurations requires identification of the "bottleneck" tier and choosing the right mechanism to alleviate it. In some cases, adaptation decisions must be made for multiple tiers. As we show in Chapter 6, efficient hosting of Web application often requires careful selection of the best adaptation mechanism to be applied at more than one tier. Moreover, the adaptation decisions must be based on the end-to-end performance of the Web application instead of adapting individual tiers independently. We address the issue of adaptation of Web applications in detail in Chapter 6.

2.5. REPLICA PLACEMENT

The task of replica placement algorithms is to find good locations to host replicas. As noted earlier, replica placement forms a controllable input parameter of the objective function. Changes in uncontrollable parameters, such as client request

rate or client latencies, may warrant changing the replica locations. In such case, the adaptation triggering component triggers the replica placement algorithms, which subsequently adapt the current placement to new conditions.

The problem of replica placement consists of two subproblems: *replica server placement* and *replica content placement*. Replica server placement is the problem of finding suitable locations for replica servers. Replica content placement is the problem of selecting replica servers that should host replicas of an object. Both these placements can be adjusted by the system to optimize the objective function value λ.

There are some fundamental differences between the server and content placement problems. Server placement concerns the selection of locations that are good for hosting replicas of *many* objects, whereas content placement deals with the selection of locations that are good for replicas of a *single* object. Furthermore, these two problems differ in how often and by whom their respective solutions need to be applied. The server placement algorithms are used in a larger time scale than the content placement algorithms. They are usually used by the system operator during installation of server infrastructure or while upgrading the hosting infrastructure, and can typically be run once every few months. The content placement algorithms are run more often, as they need to react to possibly rapidly changing situations such as flash crowds.

In [Karlsson et al., 2002], the authors present a framework for evaluating replica placement algorithms for content delivery networks and also in other fields such as distributed file systems and databases. Their framework can be used to classify and qualitatively compare the performance of various algorithms using a generic set of primitives covering problem definition and heuristics. They also provide an analytical model to predict the decision times of each algorithm. Their framework is useful for evaluating the *relative performance* of different replica placement algorithms, and as such, complements the material discussed in this section.

2.5.1. Replica server placement

The problem of replica server placement is to select K servers out of N potential sites such that the objective function is optimized for a given network topology, client population, and access patterns. The objective function used by the server placement algorithms operates on some of the metrics defined in Section 2.3. These metrics may include, for example, client latencies for the objects hosted by the system, and the financial cost of server infrastructure.

The problem of determining the number and locations of replica servers, given a network topology, can be modeled as the center placement problem. Two variants used for modeling it are the facility location problem and the minimum K-

median problem. Both these problems are NP-hard. They are defined in [Shmoys et al., 1997; Qiu et al., 2001], and we describe them here again for the sake of completeness.

Facility Location Problem Given a set of candidate server locations i in which the replica servers ("facilities") may be installed, running a server in location i incurs a cost of f_i. Each client j must be assigned to one replica server, incurring a cost of $d_j c_{ij}$ where d_j denotes the demand of the node j, and c_{ij} denotes the distance between i and j. The objective is to find the number and location of replica servers which minimizes the overall cost.

Minimum K-Median Problem Given N candidate server locations, we must select K of them (called "centers"), and then assign each client j to its closest center. A client j assigned to a center i incurs a cost of $d_j c_{ij}$. The goal is to select K centers, so that the overall cost is minimal.

The difference between the minimum K-median problem and the facility location problem is that the former associates no cost with opening a center (as with a facility, which has an operating cost of f_i). Further, in the minimum K-median problem, the number of servers is bounded by K.

Some initial work on the problem of replica server placement has been addressed in [da Cunha, 1997]. However, it has otherwise been seldom addressed by the research community and only few solutions have been proposed.

In [Li et al., 1999], the authors propose a placement algorithm based on the assumption that the underlying network topologies are trees and solve it using dynamic programming techniques. The algorithm is designed for Web proxy placement but is also relevant to server placement. The algorithm works by dividing a tree T into smaller subtrees T_i; the authors show that the best way to place t proxies is by placing t_i proxies for each T_i such that $\sum t_i = t$. The algorithm is shown to be optimal if the underlying network topology is a tree. However, this algorithm has two limitations: (i) it cannot be applied to a wide-area network such as the Internet whose topology is not a tree, and (ii) it has a high computational complexity of $O(N^3K^2)$ where K is the number of proxies and N is the number of candidate locations. We note that the first limitation of this algorithm is due to its assumption about the presence of a single origin server and henceforth to find servers that can host a target Web service. This allows to construct only a tree topology with this origin server as root. However, a typical Web replica hosting system will host documents from multiple origins, falsifying this assumption. This nature of problem formulation is more relevant for content placement, where every document has a single origin Web server.

In [Qiu et al., 2001], the authors model the replica placement problem as a minimum K-median problem and propose a greedy algorithm. In each iteration,

the algorithm selects one server, which offers the least cost, where cost is defined as the average distance between the server and its clients. In the i^{th} iteration, the algorithm evaluates the cost of hosting a replica at the remaining $N-i+1$ potential sites in the presence of already selected $i-1$ servers. The computational cost of the algorithm is $O(N^2K)$. The authors also present a hot-spot algorithm, in which the replicas are placed close to the clients generating most requests. The computational complexity of the hot-spot algorithm is $N^2 + min(NlogN, NK)$. The authors evaluate the performance of these two algorithms and compare each one with the algorithm proposed in [Li et al., 1999]. Their analysis shows that the greedy algorithm performs better than the other two algorithms and its performance is only 1.1 to 1.5 times worse than the optimal solution. The authors note that the placement algorithms need to incorporate the client topology information and access pattern information, such as client end-to-end distances and request rates.

In [Radoslavov et al., 2001], the authors propose two replica server placement algorithms that do not require the knowledge of client location but decide on replica location based on the network topology alone. The proposed algorithms are *max router fanout* and *max AS/max router fanout*. The first algorithm selects servers closest to the router having maximum fanout in the network. The second algorithm first selects the Autonomous System (AS) with the highest fanout, and then selects a server within that AS that is closest to the router having maximum fanout. The performance studies show that the second algorithm performs only 1.1 to 1.2 times worse than that of the greedy algorithm proposed in [Qiu et al., 2001]. Based on this, the authors argue that the need for knowledge of client locations is not essential. However, it must be noted that these topology-aware algorithms assume that the clients are uniformly spread throughout the network, which may not be true. If clients are not spread uniformly throughout the network, then the algorithm can select replica servers that are close to routers with highest fanout but distant from most of the clients, resulting in poor client-perceived performance.

2.5.2. Replica content placement

The problem of replica content placement consists of two subproblems: *content placement* and *replica creation*. The first problem concerns the selection of a set of replica servers that must hold the replica of a given object. The second problem concerns the selection of a mechanism to inform a replica server about the creation of new replicas.

Content placement

The content placement problem consists of selecting K out of N replica servers to host replicas of an object, such that the objective function is optimized under

a given client access pattern and replica update pattern. The content placement algorithms select replica servers in an effort to improve the quality of service provided to the clients and minimize the object hosting cost.

Similarly to the server placement, the content placement problem can be modeled as the facility location placement. However, such solutions can be computationally expensive making it difficult to be applied to this problem, as the content placement algorithms are run far more often their server-related counterparts. Therefore, existing replica hosting systems exploit simpler solutions.

In Radar [Rabinovich and Aggarwal, 1999], every host runs the replica placement algorithm, which defines two client request rate thresholds: R_{rep} for replica replication, and R_{del} for object deletion, where $R_{del} < R_{rep}$. A document is deleted if its client request rate drops below R_{del}. The document is replicated if its client request rate exceeds R_{rep}. For request rates falling between R_{del} and R_{rep}, documents are migrated to a replica server located closer to clients that issue more than a half of requests. The distance is calculated using a Radar-specific metric called preference paths. These preference paths are computed by the servers based on information periodically extracted from routers.

In SPREAD, the replica servers periodically calculate the expected number of requests for an object. Servers decide to create a local replica if the number of requests exceeds a certain threshold [Rodriguez and Sibal, 2000]. These servers remove a replica if the popularity of the object decreases. If required, the total number of replicas of an object can be restricted by the object owner.

In [Chen et al., 2002a], the authors propose a dynamic replica placement algorithm for scalable content delivery. This algorithm uses a *dissemination-tree*-based infrastructure for content delivery and a peer-to-peer location service provided by Tapestry for locating objects [Zhao et al., 2004]. The algorithm works as follows. It first organizes the replica servers holding replicas of the same object into a load-balanced tree. Then, it starts receiving client requests which target the origin server containing some latency constraints. The origin server services the client if the server's capacity constraints and client's latency constraints are met. If any of these conditions fail, it searches for another server in the dissemination tree that satisfies these two constraints and creates a replica at that server. The algorithm aims to achieve better scalability by quickly locating the objects using the peer-to-peer location service. The algorithm is good in terms of preserving client latency and server capacity constraints. On the other hand, it has a considerable overhead caused by checking QoS requirements for every client request. In the worst case a single client request may result in creating a new replica. This can significantly increase the request servicing time.

In [Kangasharju et al., 2001a], the authors model the content placement problem as an optimization problem. The problem is to place K objects in some of N

servers, in an effort to minimize the average number of inter-AS hops a request must traverse to be serviced, meeting the storage constraints of each server. The problem is shown to be NP-complete and three heuristics are proposed to address this problem. The first heuristic uses popularity of an object as the only criterion and every server decides upon the objects it needs to host based on the objects' popularity. The second heuristic uses a cost function defined as a product of object popularity and distance of server from origin server. In this heuristic, each server selects the objects to host as the ones with high cost. The intuition behind this heuristic is that each server hosts objects that are highly popular and also that are far away from their origin server, in an effort to minimize the client latency. This heuristic always tries to minimize the distance of a replica from its origin server oblivious of the presence of other replicas. The third heuristic overcomes this limitation and uses a coordinated replication strategy where replica locations are decided in a global/coordinated fashion for all objects. This heuristic uses a cost function that is a product of total request rate for a server, popularity, and shortest distance of a server to a copy of the object. The central server selects the object and replica pairs that yield the best cost at every iteration and recomputes the shortest distance between servers for each object. Using simulations, the authors show that the global heuristic outperforms the other two heuristics. The drawback is its high computational complexity.

In [Tang and Xu, 2004], the authors model the content placement problem as a constrained optimization problem where the objective is to find a placement position such that the CDN can minimize its replication costs (e.g., bandwidth costs, or storage) provided it can meet certain a quality of service (e.g., client latency). The authors show that the problem is NP-complete and propose heuristics that perform close to optimal solution and has $O(N^2)$ complexity.

In [Szymaniak et al., 2006], the authors propose a latency-minimizing content placement algorithm that is designed to scale for millions of servers. The algorithm works in two steps. The first step is to identify the *network regions* (i.e., group of nodes whose latencies to each other are relatively low), where the replicas should be placed. The second step is to choose a server to host the replica in the selected network region. The algorithm relies on network positioning techniques that model Internet latencies in an M-dimensional geometric space (discussed in Section 2.3). This allows the system to avoid the costly pairwise latency estimations between all potential replica locations. This reduces the computational cost of choosing K replica locations out of N servers to $O(Nmax(logN,K))$, which is significantly lower than those of the previously proposed algorithms.

Replica creation mechanisms

Various mechanisms can be used to inform a replica server about the creation of a new replica that it needs to host. The most widely used mechanisms for this purpose are *pull-based caching* and *push replication*.

In pull-based caching, replica servers are not explicitly informed of the creation of a new replica. When a replica server receives a request for a document it does not own, it treats it as a miss and fetches the replica from the master. As a consequence, the creation of a new replica is delayed until the first request for this replica. This scheme is adopted in Akamai [Dilley et al., 2002]. Note that in this case, pull-based caching is used only as a mechanism for replica creation. The decision to place a replica in that server is taken by the system, when redirecting client requests to replica servers.

In push replication, a replica server is informed of a replica creation by explicitly pushing the replica contents to the server. A similar scheme is used in Globule [Pierre and van Steen, 2006] and Radar [Rabinovich and Aggarwal, 1999].

2.5.3. Discussion

The problem of replica server and content placement is not regularly addressed by the research community. A few works have proposed solution for these problems [Qiu et al., 2001; Radoslavov et al., 2001; Chen et al., 2002a; Kangasharju et al., 2001a]. We note that an explicit distinction between server and content placement is generally not made. Rather, work has concentrated on finding server locations to host contents of a single content provider. However, separate solutions for server placement and content placement would be more useful in a replica hosting system, as these systems are intended to host different contents with varying client access patterns.

The existing server placement algorithms improve client QoS by minimizing client latency or distance [Qiu et al., 2001; Radoslavov et al., 2001]. Even though client QoS is important to make placement decisions, in practice the selection of replica servers is constrained by administrative reasons, such as business relationship with an ISP, and financial cost for installing a replica server. Such a situation introduces a necessary tradeoff between financial cost and performance gain, which are not directly comparable entities. This drives the need for server placement solutions that not only take into account the financial cost of a particular server facility but that can also translate the performance gains into potential monetary benefits. To the best of our knowledge little work has been done in this area, which requires building economic models that translate the performance of a replica hosting system into the monetary profit gained. These kinds of economic models are imperative to enable system designers to make better judgments in

server placement and provide server placement solutions that can be applied in practice.

Note that the content placement algorithms described in this section are designed for replicating static Web objects. However, the problem of placing replicas of Web applications is vastly different from placing Web objects. This is because unlike static Web objects a Web application is multi-tiered and replicating a Web application requires replication of (all or some of) its tiers. For instance, we can replicate the business logic code alone and use a centralized database (e.g., Akamai's Edge Computing Infrastructure [Dilley et al., 2002]). Alternatively, we can place the business logic code and the entire database at all servers. Both approaches have their own advantages and disadvantages. We return to this problem and discuss some of the significant research efforts addressing this issue in Section 2.8.

2.6. CONSISTENCY ENFORCEMENT

The consistency enforcement problem concerns selecting *consistency models* and implementing them using various *consistency policies*, which in turn can use several *content distribution mechanisms*. A consistency model is a contract between a replica hosting system and its clients that dictates the consistency-related properties of the content delivered by the system. A consistency policy defines how, when, and to which object replicas the various content distribution mechanisms are applied. For each object, a policy adheres to a certain consistency model defined for that object. A single model can be implemented using different policies. A content distribution mechanism is a method by which replica servers exchange replica updates. It defines in what form replica updates are transferred, who initiates the transfer, and when updates take place.

Although consistency models and mechanisms are usually well defined, choosing a valid one for a given object is a nontrivial task. The selection of a consistency model, policies, and mechanisms must ensure that the required level of consistency (defined by various consistency metrics as discussed in Section 2.3) is met, while keeping the communication overhead as low as possible.

2.6.1. Consistency models

Consistency models differ in their strictness of enforcing consistency. By strong consistency, we mean that the system guarantees that all replicas are identical from the perspective of the system's clients. If a given replica is not consistent with others, it cannot be accessed by clients until it is brought up to date. Due to high replica synchronization costs, strong consistency is seldom used in wide-area

systems. Weak consistency, in turn, allows replicas to differ, but ensures that all updates reach all replicas after some (bounded) time. Since this model is resistant to delays in update propagation and incurs less synchronization overhead, it fits better in wide-area systems.

Single vs. multiple master

Depending on whether updates originate from a single site or from several ones, consistency models can be classified as single-master or multi-master, respectively. The single-master models define one machine to be responsible for holding an up-to-date version of a given object. These models are simple and fit well with applications where the objects by nature have a single source of changes. They are also commonly used in existing replica hosting systems, as these systems usually deliver some centrally managed data. For example, Radar assumes that most of its objects are static Web objects that are modified rarely and uses primary-copy mechanisms for enforcing consistency. The multi-master models allow more than one server to modify the state of an object. These models are applicable to replicated Web objects whose state can be modified as a result of a client access. However, these models introduce some new problems such as the necessity of solving update conflicts. Little work has been done on these models in the context of Web replica hosting systems.

Types of consistency

As we explained, consistency models usually define consistency along three different axes: time, value, and order.

Time-based consistency models were formalized in [Torres-Rojas et al., 1999] and define consistency based on real time. These models require a content distribution mechanism to ensure that an update to a replica at time t is visible to the other replicas and clients before time $t + \Delta$. In [Cate, 1992], the author adopts a time-based consistency model for maintaining consistency of FTP caches. The consistency policy in this system guarantees that the only updates that might not yet be reflected in a site are the ones that have happened in the last 10% of the reported age of the file. Time-based consistency is applicable to all kinds of objects. It can be enforced using different content distribution mechanisms such as *polling* (where a client or replica polls often to see if there is an update), or server invalidation (where a server invalidates a copy held by other replicas and clients if it gets updated). These mechanisms are explained in detail in the next section.

Value-based consistency schemes ensure that the difference between the value of a replica and that of other replicas (and its clients) is no greater than a certain Δ. Value-based schemes can be applied only to objects that have a precise definition

of value. For example, an object encompassing the details about the number of seats booked in an aircraft can use such a model. This scheme can be implemented by using polling or server invalidation mechanisms. Examples of value-based consistency schemes and content distribution mechanisms can be found in [Bhide et al., 2002].

Order-based consistency schemes are generally exploited in replicated databases. These models perceive every read/write operation as a transaction and allow the replicas to operate in different states if the out-of-order transactions adhere to the rules defined by these policies. For example,in [Krishnakumar and Bernstein, 1994], the authors introduce the concept of N-ignorant transactions, where a transaction can be carried out on a replica while it is ignorant of N prior transactions in other replicas. The rules constraining the order of execution of transactions can also be defined based on dependencies among transactions. Implementing order-based consistency policies requires content distribution mechanisms to exchange the transactions among all replicas, and transactions need to be timestamped using mechanisms such as logical clocks [Raynal and Singhal, 1996]. This consistency scheme is applicable to a group of objects that jointly constitute a regularly updated database.

A continuous consistency model, integrating the above three schemes, is presented in [Yu and Vahdat, 2002]. The underlying premise of this model is that there is a continuum between strong and weak consistency models that is semantically meaningful for a wide range of replicated services, as opposed to traditional consistency models, which explore either strong or weak consistency [Bernstein and Goodman, 1983]. The authors explore the space between these two extremes by making applications specify their desired level of consistency using *conits*. A conit is defined as a unit of consistency. The model uses a three-dimensional vector to quantify consistency: *(Numerical Error, Order Error, Staleness)*. *Numerical error* is used to define and implement value-based consistency, *Order error* is used to define and implement order-based consistency schemes, and *Staleness* is used for time-based consistency. If each of these metrics is bound to zero, then the model implements strong consistency. Similarly, if there are no bounds then the model does not provide any consistency at all. The conit-based model allows a broad range of applications to express their consistency requirements. Also, it can precisely describe guarantees or bounds with respect to differences between replicas on a per-replica basis. This enables replicas having poor network connectivity to implement relaxed consistency, whereas replicas with better connectivity can still benefit from stronger consistency. The mechanisms implementing this conit-based model are described in [Yu and Vahdat, 2000] and [Yu and Vahdat, 2002].

2.6.2. Content distribution mechanisms

Content distribution mechanisms define how replica servers exchange updates. These mechanisms differ on two aspects: the forms of the update and the direction in which updates are triggered (from update source to replicas or vice versa). The choices made for each aspect influence the system's attainable level of consistency as well as the communication overhead incurred by consistency enforcement.

Update forms

Replica updates can be transferred in three different forms. In the first form, called *state shipping*, a whole replica is sent. The advantage of this approach is its simplicity. On the other hand, it may incur significant communication overhead, especially noticeable when a small update is performed on a large object.

In the second update form, called *delta shipping*, only differences with the previous state are transmitted. It generally incurs less communication overhead compared to state shipping, but it requires each replica server to have the previous replica version available. Further, delta shipping assumes that the differences between two object versions can be quickly computed.

In the third update form, called *function shipping*, replica servers exchange the actions that cause the changes. It generally incurs the least communication overhead as the size of description of the action is usually independent from the object state and size. However, it forces each replica server to convey a certain, possibly computationally demanding, operation.

The update form is usually dictated by the exploited replication scheme and the object characteristics. For example, in *active replication* requests targeting an object are processed by all the replicas of this object. In such a case, function shipping is the only choice. In *passive replication*, in turn, requests are first processed by a single replica, and then the remaining ones are brought up-to-date. In such a case, the update form selection depends on the object characteristics and the change itself: whether the object structure allows for changes to be easily expressed as an operation (which suggests function shipping), whether the object size is large compared to the size of the changed part (which suggests delta shipping), and finally, whether the object was simply replaced with a completely new version (which suggests state shipping).

In general, it is the job of a system designer to select the update form that minimizes the overall communication overhead. In most replica hosting systems, updating means simply replacing the whole replica with its new version. However, it has been shown that updating Web objects using delta shipping could reduce the communication overhead by up to 22% compared to commonly used state shipping [Mogul et al., 1997].

Update direction

The update transfer can be initiated either by a replica server that is in need of a new version and wants to *pull* it from one of its peers, or by the replica server that holds a new replica version and wants to *push* it to its peers. It is also possible to combine both mechanisms.

Pull In one version of the pull-based approach, every piece of data is associated with a *Time To Refresh* (TTR) attribute, which denotes the next time the data should be validated. The value of TTR can be a constant, or can be calculated from the update rate of the data. It may also depend on the consistency requirements of the system. Data with high update rates and strong consistency requirements require a small TTR, whereas data with less updates can have a large TTR. Such a mechanism is used in [Cate, 1992]. The advantage of the pull-based scheme is that it does not require replica servers to store state information, offering the benefit of higher fault tolerance. On the other hand, enforcing stricter consistency depends on careful estimation of TTR: small TTR values will provide good consistency, but at the cost of unnecessary transfers when the document was not updated.

In another pull-based approach, HTTP requests targeting an object are extended with the HTTP *if-modified-since* field. This field contains the modification date of a cached copy of the object. Upon receiving such a request, a Web server compares this date with the modification date of the original object. If the Web server holds a newer version, the entire object is sent as the response. Otherwise, only a header is sent, notifying that the cached copy is still valid. This approach allows for implementing strong consistency. On the other hand, it can impose large communication overhead, as the object home server has to be contacted for each request, even if the cached copy is valid.

In practice, a combination of TTR and checking the validity of a document at the server is used. Only after the TTR value expires, will the server contact the document's origin server to see whether the cached copy is still valid. If it is still valid, a fresh TTR value is assigned to it and a next validation check is postponed until the TTR value expires again.

Push The push-based scheme ensures that communication occurs only when there is an update. The key advantage of this approach is that it can meet strong consistency requirements without introducing the communication overhead known from the "if-modified-since" approach: since the replica server that initiates the update transfer is aware of changes, it can precisely determine which changes to push and when. An important constraint of the push-based scheme is that the

object home server needs to keep track of all replica servers to be informed. Although storing this list may seem costly, it has been shown that it can be done in an efficient way [Cao and Liu, 1998]. A more important problem is that the replica holding the state becomes a potential single point of failure, as the failure of this replica affects the consistency of the system until it is fully recovered.

Push-based content distribution schemes can be associated with leases [Gray and Cheriton, 1989]. In such approaches, a replica server registers its interest in a particular object for an associated lease time. The replica server remains registered at the object home server until the lease time expires. During the lease time, the object home server pushes all updates of the object to the replica server. When the lease expires, the replica server can either consider it as potentially stale or register at the object home server again.

Leases can be divided into three groups: age-based, renewal-frequency-based, and load-based ones [Duvvuri et al., 2000]. In the age-based leases, the lease time depends on the last time the object was modified. The underlying assumption is that objects that have not been modified for a long time will remain unmodified for some time to come. In the renewal-frequency-based leases, the object home server gives longer leases to replica servers that ask for replica validation more often. In this way, the object server prefers replica servers used by clients expressing more interest in the object. Finally, in the load-based leases the object home server tends to give away shorter lease times when it becomes overloaded. By doing that, the object home server reduces the number of replica servers to which the object updates have to be pushed, which is expected to reduce the size of the state held at the object home server.

Other schemes The pull and push approaches can be combined in different ways. In [Bhide et al., 2002], the authors propose three different combination schemes of Push and Pull. The first scheme, called *Push-and-Pull (PaP)*, simultaneously employs push and pull to exchange updates and has tunable parameters to control the extent of push and pulls. The second scheme, *Push-or-Pull (PoP)*, allows a server to adaptively choose between a push- or pull-based approach for each connection. This scheme allows a server to characterize which clients (other replica servers or proxies to which updates need to be propagated) should use either of these two approaches. The characterization can be based on system dynamics. By default, clients are forced to use the pull-based approach. PoP is a more effective solution than PaP, as the server can determine the moment of switching between push and pull, depending on its resource availability. The third scheme, called *PoPoPaP*, is an extended version of PoP, that chooses from Push, Pull and PaP. PoPoPaP improves the resilience of the server (compared to PoP),

offers graceful degradation, and can maintain strong consistency.

Another way of combining push and pull is to allow the former to trigger the latter. It can be done either explicitly, by means of *invalidations*, or implicitly, with *versioning*. Invalidations are pushed by an object's origin server to a replica server. They inform the replica server or the clients that the replica it holds is outdated. In case the replica server needs the current version, it pulls it from the origin server. Invalidations may reduce the network overhead, compared to pushing regular updates, as the replica servers do not have to hold the current version for all the time and can delay its retrieval until it is really needed. It is particularly useful for often-updated, rarely-accessed objects.

Versioning techniques are exploited in Akamai [Dilley et al., 2002; Leighton and Lewin, 2000]. In this approach, every object is assigned a unique version identifier, modified after each update. The parent document that contains a reference to the object is rewritten after each update as well, so that it points to the latest version. The consistency problem is thus reduced to maintaining the consistency of the parent document. Each time a client retrieves the document, the object reference is followed and a replica server is queried for that object. If the replica server notices that it does not have a copy of the referenced version, the new version is pulled in from the origin server.

Scalable mechanisms All the aforesaid content distribution mechanisms do not scale for large number of replicas (say, in the order of thousands). In this case, push-based mechanisms suffer from the overhead of storing the state of each replica and updating them (through separate unicast connections). Pull-based mechanisms suffer from the disadvantage of creating a hot spot around the origin server with thousands of replicas requesting the origin server (again through separate connections) for an update periodically. Both mechanisms suffer from excessive network traffic for updating large number of replicas, as the same updates are sent to different replicas using separate connections. This also introduces considerable overhead on the server, in addition to increasing the network overhead. These scalability limitations require the need for building scalable mechanisms.

Scalable content distribution mechanisms proposed in the literature aim to solve scalability problems of conventional push and pull mechanisms by building a content distribution hierarchy of replicas or clustering objects.

The first approach is adopted in [Ninan et al., 2002; Tewari et al., 2002; Fei, 2001]. In this approach, a content distribution tree of replicas is built for each object. The origin server sends its update only to the root of the tree (instead of the entire set of replicas), which in turn forwards the update to the next level of nodes in the tree and so on. The content distribution tree can be built either using network multicasting or application-level multicasting solutions. This approach drastically

reduces the overall amount of data shipped by the origin server. In [Ninan et al., 2002], the authors proposed a scalable lease-based consistency mechanism where leases are made with a replica group (with the same consistency requirement), instead of individual replicas. Each lease group has its own content distribution hierarchy to send their replica updates. Similarly, in [Tewari et al., 2002], the authors propose a mechanism that builds a content distribution hierarchy and also uses object clustering to improve the scalability.

In [Fei, 2001], the authors propose a mechanism that chooses between update propagation (through a multicast tree) or invalidation schemes on a per-object basis, periodically, based on each object's update and access rate. The basic intuition of the mechanism is to choose propagation if an object is accessed more than it is updated (thereby reducing the pull traffic) and invalidation otherwise (as the overhead for shipping updates is higher than pulling updates of an object only when it is accessed). The mechanism computes the traffic overhead of the two methods for maintaining consistency of an object, given its past update and access rate. It chooses the one that introduces the least overhead as the mechanism to be adopted for that object.

Object clustering is the process of clustering various objects with similar properties (update and/or request patterns) and treating them as a single clustered object. It reduces the connection initiation overhead during the transmission of replica updates, from a per-object level to per-cluster level, as updates for a cluster are sent in a single connection instead of individuals connection for each object (note the amount of updates transferred using both mechanisms are the same). Clustering also reduces the number of objects to be maintained by a server, which can help in reducing the adaptation overhead as a single decision will affect more objects. To our knowledge, object clustering is not used in any well-known replica hosting system.

2.6.3. Discussion

In this section, we discussed two important components of consistency enforcement namely, consistency models and content distribution mechanisms. In consistency models, we listed different types of consistency models – based on time, value or transaction orders. In addition to these models, we also discussed the continuous consistency model, in which different network applications can express the consistency constraints in any point in the consistency spectrum. This model is useful to capture the consistency requirements for a broad range of applications being hosted by a replica hosting system. However, the mechanisms proposed to enforce its policies do not scale with increasing number of replicas. Similar models need to be developed for Web replica hosting systems that can provide bounds on inconsistent access of its replicas with no loss of scalability.

Table 2.2: A comparison of approaches for enforcing consistency.

Systems and Protocols	Push		Pull	Variants	Comments
	Inv.	Prop.			
Akamai	X		X		Uses push-based invalidation for consistency and pull for distribution
Radar		X			Uses primary-copy
SPREAD				X	Chooses strategy on a per-object basis based on its access and update rate
[Pierre et al., 2002]				X	Chooses strategy on a per-object basis based on its access and update rate
[Duvvuri et al., 2000]	X				Invalidates content until lease is valid
Adaptive Push-Pull		X	X		Chooses between push and pull strategy on a per-object basis
[Fei, 2001]	X	X			Chooses between propagation and invalidation on a per-object basis

In content distribution mechanisms, we discussed the advantages and disadvantages of push, pull, and other adaptive mechanisms. These mechanisms can be broadly classified as server-driven and client-driven consistency mechanisms, depending on who is responsible for enforcing consistency. At the outset, client-driven mechanisms seems to be a more scalable option for large-scale hosting systems, as in this case the server is not overloaded with the responsibility of enforcing consistency. However, in [Yin et al., 2002] the authors have shown that server-driven consistency protocols can meet the scalability requirements of large-scale dynamic Web services delivering both static and dynamic Web content.

We note that existing systems and protocols concentrate only on time-based consistency models and very little has been done on other consistency models. Hence, in our summary table of consistency approaches adopted by various systems and protocols (Table 2.2), we discuss only the content distribution mechanisms adopted by them.

Note that the consistency mechanisms presented in this section apply mostly to static Web objects rather than relational databases. This is because, in the case of static Web objects, the read and update requests are always addressed to a single object and the ratio of updates are usually low. In the case of relational database driven Web applications, a single database query might require accessing millions of records to generate a response. Hence, the problem of providing a consistent response to a single query ensuring tracking updates to millions of objects thereby making it harder. We return to this issue in Section 2.8. The issue of data consistency is one of the key problems to solve while distributing a Web application at

a worldwide scale and is also addressed in the solutions presented in subsequent chapters.

Note that most of the consistency mechanisms described in this chapter (and those used in the systems proposed in this thesis) assume that servers and networks are failure free. However, in reality, these failures happen. An ideal replica hosting system must be capable of serving consistent data even amidst server and network failures. In [Brewer, 2000], the author conjectures that it is impossible to provide both strong consistency and perfect availability in a system that is prone to network partitions. This conjecture was proved by [Gilbert and Lynch, 2002]. These results suggest that the hosting system (or the application designer) has to choose the right tradeoff between consistency and availability. Unfortunately, most of the contributions presented in this thesis do not focus on availability-related issues. However, we occasionally return to these issues in subsequent chapters.

2.7. REQUEST ROUTING

In request routing, we address the problem of deciding which replica server shall best service a given client request, in terms of the metrics selected in Section 2.3. These metrics can be, for example, replica server load (where we choose the replica server with the lowest load), end-to-end latency (where we choose the replica server that offers the shortest response time to the client), or distance (where we choose the replica server that is closest to the client).

Selecting a replica is difficult, because the conditions on the replica servers (e.g., load) and in the network (e.g., link congestion, thus its latency) change continuously. These changing conditions may lead to different replica selections, depending on when and for which client these selections are made. In other words, a replica optimal for a given client may not necessarily remain optimal for the same client forever. Similarly, even if two clients request the same document simultaneously, they may be directed to different replicas. In this section, we refer to these two kinds of conditions as "system conditions."

The entire request routing problem can be split into two: devising a redirection policy and selecting a redirection mechanism. A redirection policy defines how to select a replica in response to a given client request. It is basically an algorithm invoked when the client request is invoked. A redirection mechanism, in turn, is a means of informing the client about this selection. It first invokes a redirection policy, and then provides the client with the redirecting response that the policy generates.

A redirection system can be deployed either on the client side, or on the server side, or somewhere in the network between these two. It is also possible to com-

bine client-side and server-side techniques to achieve better performance [Karaul et al., 1998]. Interestingly, a study by [Rodriguez et al., 2000] suggests that clients may easily circumvent the problem of replica selection by simultaneously retrieving their data from several replica servers. This claim is disputed by [Kangasharju et al., 2001b], where the authors notice that the delay caused by opening connections to multiple servers can outweigh the actual gain in content download time. In this section, we assume that we leave the client-side unmodified, as the only software that usually works there is a Web browser. We therefore do not discuss the details of client-side server-selection techniques, which can be found in [Conti et al., 2002]. Finally, we do not discuss various Web caching schemes, which have been thoroughly described in [Rodriguez et al., 2001], as caches are by nature deployed on the client-side.

In this section, we examine redirection policies and redirection mechanisms separately. For each of them, we discuss several related research efforts, and summarize with a comparison of these efforts.

2.7.1. Redirection policies

A redirection policy can be either adaptive or non-adaptive. The former considers current system conditions while selecting a replica, whereas the latter does not. Adaptive redirection policies are usually more complex than non-adaptive ones, but this effort is likely to pay off with higher system performance. The systems we discuss below usually implement both types of policies and can be configured to use any combination of them.

Non-adaptive policies

Non-adaptive redirection policies select a replica that a client should access without monitoring the current system conditions. Instead, they exploit heuristics that assume certain properties of these conditions of which we discuss examples below. Although non-adaptive policies are usually easier to implement, the system works efficiently only when the assumptions made by the heuristics are met.

An example of a non-adaptive policy is round-robin. It aims at balancing the load of replica servers by evenly distributing all the requests among these servers [Delgadillo, 1999; Radware, 2002; Szymaniak et al., 2003]. The assumption here is that all the replica servers have similar processing capabilities, and that any of them can service any client request. This simple policy has proved to work well in clusters, where all the replica servers are located in the same place [Pai et al., 1998]. In wide-area systems, however, replica servers are usually distant from each other. Since round-robin ignores this aspect, it cannot prevent directing client requests to more distant replica servers. If that happens, the client-perceived

performance may turn out to be poor. Another problem is that the aim of load balancing itself is not necessarily achieved, as processing different requests can involve significantly different computational costs.

A non-adaptive policy exploited in Radar is the following. All replica servers are ranked according to their predicted load, which is derived from the number of requests each of them has serviced so far. Then, the policy redirects clients so that the load is balanced across the replica servers, and that (additionally) the client-server distance is as low as possible. The assumption here is that the replica server load and the client-server distance are the main factors influencing the efficiency of request processing. In [Aggarwal and Rabinovich, 1998], the authors observe that this simple policy often performs nearly as good as its adaptive counterpart, which we describe below. However, as both of them ignore network congestion, the resulting client-perceived performance may still turn out to be poor.

Several interesting non-adaptive policies were implemented in Cisco DistributedDirector [Delgadillo, 1999]. The first one defines the percentage of all requests that each replica server receives. In this way, it can send more requests to more powerful replica servers and achieve better resource utilization. Another policy allows for defining preferences of one replica server over the other. It may be used to temporarily relieve a replica server from service (for maintenance purposes, for example), and delegate the requests it would normally service to another server. Finally, DistributedDirector enables random request redirection, which can be used for comparisons during some system efficiency tests. Although all these policies are easy to implement, they completely ignore current system conditions, making them inadequate to react to emergency situations.

One can imagine a non-adaptive redirection policy that statically assigns clients to replicas based on their geographical location. This time, the underlying assumptions are that the clients are evenly distributed over the world, and that the geographical distance to a server reflects the network latency to that server. Although the former assumption is not likely to be valid in a general case, the latter has been verified positively as we discussed earlier. According to [Huffaker et al., 2002], the correlation between the geographical distance and the network latency reaches up to 75%. Still, since this policy ignores the load of replica servers, it can redirect clients to overloaded replica servers, which may lead to substantially degraded client experience.

Adaptive policies

Adaptive redirection policies discover the current system conditions by means of metric estimation mechanisms discussed in Section 2.3. In this way, they are able to adjust their behavior to situations that normally do not occur, like flash crowds, and ensure high system robustness [Wang et al., 2002].

The information that adaptive policies obtain from metric estimation mechanisms may include, for example, the load of replica servers or the congestion of selected network links. Apart from these data, a policy may also need to know some request-related information. The bare minimum is what object is requested and where the client is located. More advanced replica selection can also take client QoS requirements into account.

Knowing the system conditions and the client-related information, adaptive policies first determine a set of replica servers that are capable of handling the request (i.e., they store a replica of the document and can offer required quality of service). Then, these policies select one (or more) of these servers, according to the metrics they exploit. Adaptive policies may exploit more than one metric. More importantly, a selection based on one metric is not necessarily optimal in terms of others. For example, [Johnson et al., 2001] observed that most CDNs do not always select the replica server closest to the client.

The adaptive policy used by Globule selects the replica servers that are closest to the client in terms of network distance [Szymaniak et al., 2003]. Globule employs the AS-path length metric, originally proposed by [McManus, 1999], and determines the distance based on a periodically refreshed, AS-based map of the Internet. Since this approach uses passive metric estimation services, it does not introduce any additional traffic to the network. We consider it to be adaptive, because the map of the Internet is periodically rebuilt, which results in (slow) adaptation to network topology changes. Unfortunately, the AS-based distance calculations, although simple to perform, are not very accurate [Huffaker et al., 2002].

A distance-based adaptive policy is also exploited by SPREAD [Rodriguez and Sibal, 2000]. In this system, routers simply intercept requests on their path toward the object home server, and redirect to a near-by replica server. Consequently, requests reach their closest replica servers, and the resulting client-server paths are shortened. This policy in a natural way adapts to changes in routing. Its biggest disadvantage is the high cost of deployment, as it requires modifying many routers.

A combined policy, considering both replica server load and client-server distance, is implemented in Radar. The policy first isolates the replica servers whose load is below a certain threshold. Then, from these servers, the client-closest one is selected. The Radar redirection policy adapts to changing replica server loads and tries to direct clients to their closest replica servers. However, by ignoring network congestion and end-to-end latencies, Radar focuses more on load balancing than on improving the client-perceived performance.

Adaptive policies based on client-server latency have been proposed by [Ardaiz et al., 2001] and [Andrews et al., 2002]. Based either on the client access logs,

or on passive server-side latency measurements, respectively, these policies redirect a client to the replica server that has recently reported the minimal latency to the client. The most important advantage of these schemes is that they exploit latency measurements, which are the best indicator of actual client experience [Huffaker et al., 2002]. On the other hand, both of them require maintaining a central database of measurements, which limits the scalability of systems that exploit these schemes.

A set of adaptive policies is supported by Web Server Director [Radware, 2002]. It monitors the number of clients and the amount of network traffic serviced by each replica server. It also takes advantage of performance metrics specific for Windows NT, which are included in the Management Information Base (MIB). Since this information is only provided in a commercial white paper, it is difficult to evaluate the efficiency of these solutions.

Another set of adaptive policies is implemented in Cisco DistributedDirector [Delgadillo, 1999]. This system supports many different metrics, including inter-AS distance, intra-AS distance, and end-to-end latency. The redirection policy can determine the replica server based on a weighted combination of these three metrics. Although this policy is clearly more flexible than a policy that uses only one metric, measuring all the metrics requires deploying an "agent" on every replica server. Also, the exploited active latency measurements introduce additional traffic to the Internet. Finally, because DistributedDirector is kept separate from the replica servers, it cannot probe their load – it can be approximated only with the non-adaptive policies discussed above.

A complex adaptive policy is used in Akamai [Dilley et al., 2002]. It considers a few additional metrics, like replica server load, the reliability of routes between the client and each of the replica servers, and the bandwidth that is currently available to a replica server. Unfortunately, the actual policy is subject to trade secret and cannot be found in the published literature. However, a recent measurement study by [Su et al., 2006] sheds more light on the effectiveness of Akamai's redirection policy. In their study, the authors analyzed the redirection strategies of Akamai and found that in most cases Akamai's redirection policy is primarily driven by network latencies. Interestingly enough, the authors also note that Akamai is usually able to detect even short-lived congestion points in the Internet and to change its redirecting decisions accordingly.

2.7.2. Redirection mechanisms

Redirection mechanisms provide clients with the information generated by the redirection policies. Redirection mechanisms can be classified according to several criteria. For example, in [Barbir et al., 2002], the authors classify redirection mechanisms into transport-level, DNS-based, and application-level ones. The

authors use the term “request routing” to refer to what we call “redirection” in this paper. Such classification is dictated by the diversity of request processing stages, where redirection can be incorporated: packet routing, name resolution, and application-specific redirection implementation.

In this section, we distinguish between transparent, non-transparent, and combined mechanisms. Transparent redirection mechanisms hide the redirection from the clients. In other words, a client cannot determine which replica server is servicing it. In non-transparent redirection mechanisms, the redirection is visible to the client, which can then explicitly refer to the replica server it is using. Combined redirection mechanisms combine two previous types. They take the best from these two types and eliminate their disadvantages.

As we only focus on wide-area systems, we do not discuss solutions that are applicable only to local environments. An example of such a solution is packet hand-off, which is thoroughly discussed in a survey of load-balancing techniques by [Cardellini et al., 1999].

Transparent mechanisms

Transparent redirection mechanisms perform client request redirection in a transparent manner. Therefore, they do not introduce explicit bounds between clients and replica servers, even if the clients store references to replicas. It is particularly important for mobile clients and for dynamically changing network environments, as in both these cases, a replica server now optimal for a given client can become suboptimal shortly later.

Several transparent redirection mechanisms are based on DNS [Delgadillo, 1999; Rabinovich and Aggarwal, 1999; Radware, 2002; Szymaniak et al., 2003]. They exploit specially modified DNS servers. When a modified DNS server receives a resolution query for a replicated service, a redirection policy is invoked to generate one or more service IP addresses, which are returned to the client. The policy chooses the replica servers based on the IP address of the query sender. In DNS-based redirection, transparency is achieved assuming that services are referred to by means of their DNS names, and not their IP addresses. The entire redirection mechanism is extremely popular, because of its simplicity and independence from the actual replicated service – as it is incorporated in the name resolution service, it can be used by any Internet application.

On the other hand, DNS-based redirection has some limitations [Shaikh et al., 2001]. The most important ones are poor client identification and coarse redirection granularity. Poor client identification is caused by the fact that a DNS query does not necessarily carry the address of the querying client. The query can pass through several DNS servers before it reaches the one that knows the answer. However, any of these DNS servers knows only the DNS server with

which it directly communicates, and not the querying client. Consequently, using DNS-based redirection mechanisms forces the system to use the clustering scheme based on local DNS servers, which was discussed in Section 2.3. The coarse redirection granularity is caused by the granularity of DNS itself: as it deals only with machine names, it can redirect based only on the part of object URL that is related to the machine name. Therefore, as long as two object URLs refer to the same machine name, they are identical for the DNS-based redirection mechanism, which makes it difficult to use different distribution schemes for different objects.

A scalable version of DNS-based redirection is implemented in Akamai. This system improves the scalability of the redirection mechanism by maintaining two groups of DNS servers: top- and low-level ones. Whereas the former share one location, the latter are scattered over several Internet data centers, and are usually accompanied by replica servers. A top-level DNS servers redirects a client query to a low-level DNS server proximal to the query sender. Then, the low-level DNS server redirects the sender to an actual replica server, usually placed in the same Internet data center. What is important, however, is that the top-to-low level redirection occurs only periodically (about once per hour) and remains valid during all that time. For this reason, the queries are usually handled by proximal low-level DNS servers, which results in short name-resolution latency. Also, because the low-level DNS servers and the replica servers share the same Internet data center, the former may have accurate system condition information about the latter. Therefore, the low-level DNS servers may quickly react to sudden changes, such as flash crowds or replica server failures.

An original transparent redirection scheme is exploited in SPREAD, which makes proxies responsible for client redirection [Rodriguez and Sibal, 2000]. SPREAD assumes the existence of a distributed infrastructure of proxies, each handling all HTTP traffic in its neighborhood. Each proxy works as follows. It inspects the HTTP-carrying IP packets and isolates these targeting replicated services. All other packets are routed traditionally. If the requested replica is not available locally, the service-related packets are forwarded to another proxy along the path toward the original service site. Otherwise, the proxy services them and generates IP packets carrying the response. The proxy rewrites source addresses in these packets, so that the client thought that the response originates from the original service site. The SPREAD scheme can be perceived as a distributed packet hand-off. It is transparent to the clients, but it requires a whole infrastructure of proxies.

Non-transparent mechanisms

Non-transparent redirection mechanisms reveal the redirection to the clients. In this way, these mechanisms introduce an explicit binding between a client and a

given replica server. On the other hand, non-transparent redirection mechanisms are easier to implement than their transparent counterparts. They also offer fine redirection granularity (per object), thus allowing for more flexible content management.

The simplest method that gives the effect of non-transparent redirection is to allow a client to choose from a list of available replica servers. This approach is called "manual redirection" and can often be found on Web services of widely-known corporations. However, since this method is entirely manual, it is of little use for replica hosting systems, which require an automated client redirection scheme.

Non-transparent redirection can be implemented by means of HTTP. It is another redirection mechanism supported by Web Server Director [Radware, 2002]. An HTTP-based mechanism can redirect clients by rewriting object URLs inside HTML documents, so that these URLs point at object replicas stored on some replica servers. It is possible to treat each object URL separately, which allows for using virtually any replica placement. The two biggest advantages of the HTTP-based redirection are flexibility and simplicity. Its biggest drawback is the lack of transparency.

Cisco DistributedDirector also supports the HTTP-based redirection, although in a different manner [Delgadillo, 1999]. Instead of rewriting URLs, this system exploits the HTTP 302 (temporary moved) response code. In this way, the redirecting machine does not need to store any service-related content – all it does is activate the redirection policy and redirect client to a replica server. On the other hand, this solution can efficiently redirect only per entire Web service, and not per object.

Combined mechanisms

It is possible to combine transparent and non-transparent redirection mechanisms in order to achieve a better result. Such approaches are followed by Akamai [Dilley et al., 2002], Cisco DistributedDirector [Radware, 2002] and Web Server Director [Delgadillo, 1999]. These systems allow to redirect clients using a "cascade" of different redirection mechanisms.

The first mechanism in the cascade is HTTP. A replica server may rewrite URLs inside an HTML document so that the URLs of different embedded objects contain different DNS names. Each DNS name identifies a group of replica servers that store a given object.

Although it is in general not recommended to scatter objects embedded in a single Web page over too many servers [Kangasharju et al., 2001b], it may be sometimes beneficial to host objects of different *types* on separate groups of replica servers. For example, as video hosting may require specialized replica

server resources, it may be reasonable to serve video streams with dedicated video servers, while providing images with other, regular ones. In such cases, video-related URLs contain a different DNS name (like "video.cdn.com") than the image-related URLs (like "images.cdn.com").

URL rewriting weakens the transparency, as the clients are able to discover that the content is retrieved from different replica servers. However, because the rewritten URLs contain DNS names that point to *groups* of replica servers, the clients are not bound to any *single* replica server. In this way, the system preserves the most important property of transparent redirection systems.

The second mechanism in the cascade is DNS. The DNS redirection system chooses the best replica server within each group by resolving the group-corresponding DNS name. In this way, the same DNS-based mechanism can be used to redirect a client to its several best replica servers, each belonging to a separate group.

By using DNS, the redirection system remains scalable, as it happens in the case of pure DNS-based mechanisms. By combining DNS with URL rewriting, however, the system may offer finer redirection granularity and thus allow for more flexible replica placement strategies.

The third mechanism in the cascade is packet hand-off. The processing capabilities of a replica server may be improved by deploying the replica server as a cluster of machines that share the same IP address. In this case, the packet-handoff is implemented locally to scatter client requests across several machines.

Similarly to pure packet-handoff techniques, this part of the redirection cascade remains transparent for the clients. However, since packet hand-off is implemented only locally, the scalability of the redirection system is maintained.

As can be observed, combining different redirection mechanisms leads to constructing a redirection system that is simultaneously fine-grained, transparent, and scalable. The only potential problem is that deploying and maintaining such a mechanism is a complex task. In practice, however, this problem turns out to be just one more task of a replica hosting system operator. The duties like maintaining a set of reliable replica servers, managing multiple replicas of many objects, and making these replicas consistent, are likely to be at similar (if not higher) level of complexity.

2.7.3. Discussion

The problem of request routing can be divided into two subproblems: devising a redirection policy and selecting a redirection mechanism. The policy decides to which replica a given client should be redirected, whereas the mechanism takes care of delivering this decision to the client.

We classify redirection policies into two groups: adaptive and non-adaptive ones. Non-adaptive policies perform well only when the system conditions do not change. If they do change, the system performance may turn out to be poor. Adaptive policies solve this problem by monitoring the system conditions and adjusting their behavior accordingly. However, they make the system more complex, as they need specialized metric estimation services. We note that all the investigated systems implement both adaptive and non-adaptive policies (see Table 2.3).

Redirection mechanisms, in turn, are classified into three groups: transparent, non-transparent, and combined ones. Transparent mechanisms can be based on DNS or packet hand-off. As can be observed in Table 2.3, DNS-based mechanisms are very popular. Among them, a particularly interesting one is the scalable DNS-based mechanism built by Akamai. As for packet hand-off, its traditional limitation to clusters can be alleviated by means of a global infrastructure, as is done in SPREAD. Non-transparent mechanisms are based on HTTP. They achieve finer redirection granularity, at the cost of introducing an explicit binding between a client and a replica server. Transparent and non-transparent mechanisms can be combined. Resulting hybrids offer fine-grained, transparent, and scalable redirection at the cost of higher complexity.

We observe that the request routing component has to cooperate with the metric estimation services to work efficiently. Consequently, the quality of the request routing component depends on the accuracy of the data provided by the metric estimation service.

Further, we note that simple, non-adaptive policies sometimes work nearly as efficient as their adaptive counterparts. Although this phenomenon may justify using only non-adaptive policies in simple systems, we do believe that monitoring system conditions is of a key value for efficient request routing in large infrastructures. Moreover, combining several different metrics in the process of replica selection may additionally improve the system performance.

Finally, we are convinced that using combined redirection mechanisms is inevitable for large-scale wide-area systems. These mechanisms offer fine-grained, transparent, and scalable redirection at the cost of higher complexity. The resulting complexity, however, is not significantly larger compared to that of other parts of a replica hosting system. Since the ability to support millions of clients can be of fundamental importance, using a combined redirection mechanism is definitely worth the effort.

Table 2.3: The comparison of representative implementations of a redirection system

System	Redirection																
	Policies											Mechanisms					
	Adaptive						Non-Adaptive					TCP		DNS		HTTP	Comb.
	DST	LAT	NLD	CPU	USR	OTH	RR	%RQ	PRF	RND	PLD	CNT	DST	1LV	2LV		
Akamai	X	X	X	X		X									X	X	X
Globule	X						X		X					X			
Radar	X			X							X			X			
SPREAD	X												X				
Cisco DD	X	X					X	X	X	X		X		X		X	X
Web Direct			X		X	X	X					X		X		X	X

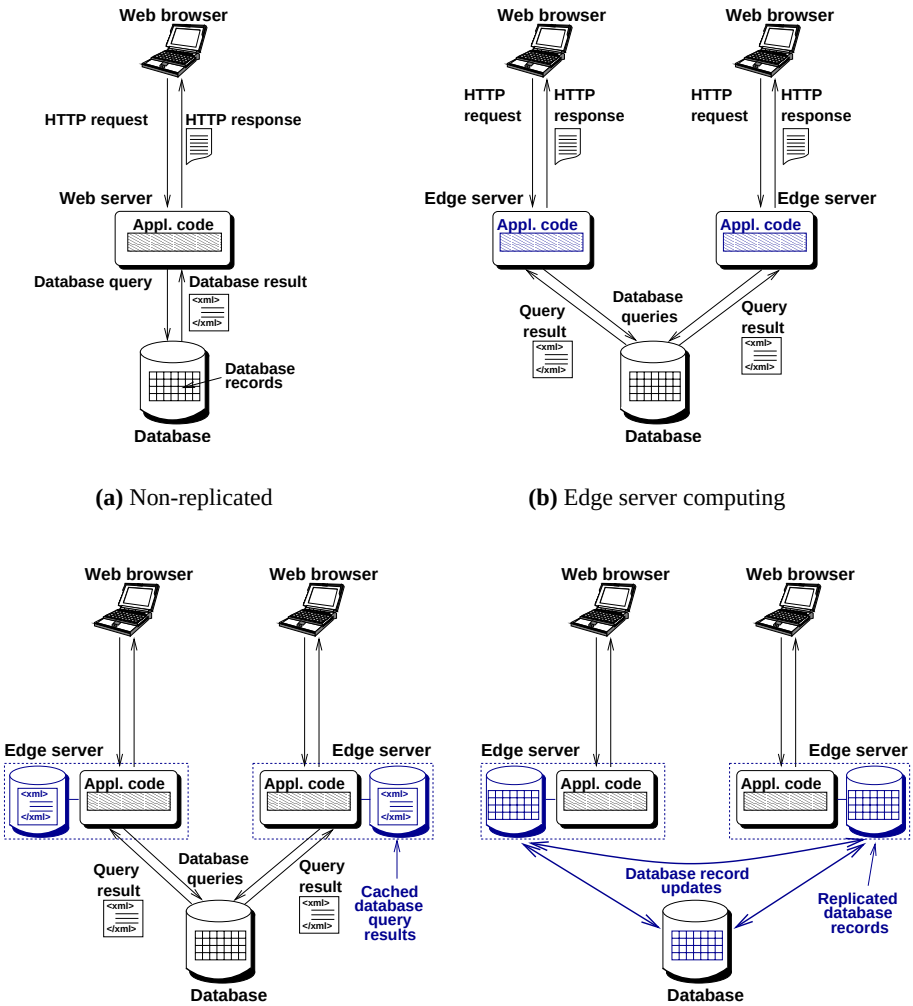
DST: Network distance
LAT : End-to-end latency
NLD: Network load
CPU: Replica server CPU load
USR: Number of users
OTH : Other metrics
RR : Round robin
%RQ: Percentage of requests
PRF : Server preference
RND : Random selection
PLD: Predicted load
CNT: Centralized
DST: Distributed
1LV : One-level
2LV : Two-level

2.8. HOSTING WEB APPLICATIONS

With the growing popularity of Web forums, e-commerce sites, blogs and many others, a significant portion of Web content is not delivered from a static file but generated dynamically each time a request is received. Dynamically generating Web contents allows servers to deliver personalized contents to each user, and to take action when specific requests are issued, such as ordering an item from an e-commerce site.

Dynamic Web applications are often organized along a three-tiered architecture, as depicted in Figure 2.7(a). When a request is issued, the Web server invokes application-specific code, which generates the content to be delivered to the client. This application code, in turn, issues queries to a database where the application state is preserved.

As noted in the previous section, most of the content placement and consistency mechanisms discussed so far are designed for static Web objects. Contrary to static Web objects, a Web application is multi-tiered and replicating Web application requires replicating (some or all of) its tiers [Sivasubramanian et al., 2006a]. In this section, we present some popular replication techniques that can be used to scale Web applications.



(a) Non-replicated

(b) Edge server computing

(c) Database query caching

(d) Database record replication

Figure 2.7: Various Web application hosting techniques.

2.8.1. Fragment caching

For hosting database-driven Web sites, one can ignore the fact that responses are generated dynamically and cache the generated responses. This is the principal idea behind a technique called fragment caching [Challenger et al., 2005; Datta et al., 2002; Douglis et al., 1997; Li et al., 2003; Rabinovich et al., 1997]. In fragment caching, each Web page is broken down into fragments, which usually consists of static data (and sometimes also certain dynamic parts of the response). The fragments are cached at edge servers and are used by the edge servers to reconstruct (parts of) the page responses. The consistency of the cached fragments is preserved either by maintaining dependency graph between the fragments and the underlying database (e.g., [Challenger et al., 2005]) or by using time-based weak consistency mechanisms (e.g., Akamai [Dilley et al., 2002]).

Fragment caching performs well for applications whose workloads exhibit high temporal locality and low update rates (to the database). These workloads tend to exhibit a good cache hit rate thereby enabling the edge servers to construct a significant fraction of the client responses at the edges [Challenger et al., 2005]. However, with growing drive towards personalization of Web sites, the response generated for each user tends to be unique, thereby reducing the benefits of these techniques. In particular, for Web applications with poor temporal locality of requests and considerable update rates, fragment caching will incur a poor cache hit rate thereby requiring most of the requests to be served by the application located in the central server.

2.8.2. Edge computing

A simple way to generate user-specific pages is to replicate the application code at multiple edge servers, and keep the data centralized (see Figure 2.7(b)). This technique is, for example, the heart of Akamai's Edge Computing [Davis et al., 2004], ACDN [Rabinovich et al., 2003] and AutoGlobe [Seltzsam et al., 2006]. Edge computing (EC) allows each edge server to generate user-specific pages according to context, sessions, and information stored in the database, thereby spreading the computational load across multiple servers.

In an EC infrastructure, one must ensure that the application code is sufficiently replicated to meet the user's demands. This is essential for many interactive Internet applications that are computationally intensive (e.g., online gaming systems). In [Rabinovich et al., 2003], the authors propose a code placement algorithm that improves upon the placement algorithm used in RADAR [Rabinovich and Aggarwal, 1999]. Compared to RADAR, ACDN's placement algorithm takes into account not only the client latency but also the bandwidth incurred in the process of code replication and migration. ACDN also replicates the application data

provided the database is never updated by the application.

Similar to ACDN, Autoglobe also proposes to adapt the number of code replicas based on the system load [Seltzsam et al., 2006]. Autoglobe uses a fuzzy logic controller to determine the best action to carry out when the application's load exceeds certain thresholds. In the event of an overload (or underload) situation, the controller identifies appropriate actions to remedy the situation. After that, the fuzzy controller calculates the applicability of all actions (e.g. replication, migration, deletion). If required, for example, for a migration action, the fuzzy controller calculates the score of all suitable target service hosts. The action with the highest applicability is executed and the host with the highest score is selected as target host of the action.

While EC infrastructures are becoming increasingly popular among commercial CDNs, the centralization of the data can pose a number of problems. First, if edge servers are located worldwide, then each data access incurs wide-area network latency; second, the central database quickly becomes a performance bottleneck as it needs to serve all database requests from the whole system. These properties restrict the use of EC to Web applications that require relatively few database accesses to generate the content.

2.8.3. Database caching

Database caching is a promising technique that can be used to alleviate the database bottleneck in edge server computing. Database caching allows data stored in the database, to be cached at locations closer to the client. This allows edge servers to answer queries locally when all the required data are present (see Figure 2.7(c)). If the data are not present, the query is forwarded to the central database.

In general, we can classify existing database caching systems into two categories based on the nature of the cached data. The first category contains systems that cache complete or partial tables from the central database server, whereas systems in the second category cache the result of executed queries.

The first category of systems that cache complete or partial tables implement this functionality by using materialized views. A materialized view consists of all the tuples that are represented by a database view [Ullman, 1990]. To maintain consistency, caches register to an update stream from the database server for the cached materialized views. Updates to the cached views will then be forwarded to the caches by the central database server. Queries that only use data which is available in the local materialized views can be answered by the cache. If the required data are not present in any locally cached view, the query is forwarded to the database server. Update, Delete and Insert (UDI) queries are always forwarded to the database server, where they will execute and possibly trigger an update to the caches.

Prominent systems that follow this approach include DBCache [Bornhövd et al., 2004], and MTCache [Larson et al., 2004]. Caching complete or partial tables is a simple, yet powerful mechanism. However, there are two scalability issues. The first one is that the materialized views are defined by the system administrator at system initialization time. This prohibits the cache to dynamically adapt to changing workloads, which can result in poor hit ratios over time. The second issue is that the granularity of the cached data can be very coarse. In that case, the amount of update traffic will therefore be considerable and a significant fraction of the cached data may not be used to answer incoming queries.

The second category of query caching systems simply caches the result of database queries as they are issued by the application code. In this case, each edge server maintains a partial copy of the database. Each time a query is issued, the edge-server database needs to check if it contains enough data locally to answer the query correctly. This process is called query containment check. If the containment check result is positive, then the query can be executed locally. Otherwise, it must be sent to the central database. In the latter case, the result is inserted in the edge-server database so that future identical requests can be served locally. The process of inserting cached tuples into the edge database is done by creating insert/update queries on the fly and requires a good understanding of the applications data schema. The update queries are always executed at the origin server (the central database server). When an edge server caches a query, it subscribes to receive invalidations of conflicting query templates. Examples of systems that adopt this approach include [Dar et al., 1996], [Amza et al., 2005], [Luo and Naughton, 2001] and DBProxy [Amiri et al., 2003a],.

Note that query caching can be done at different locations to achieve different objectives. For instance, caches placed at edge servers will help in reducing database query latency provided the query workload exhibits good temporal locality. However, keeping the cache consistent can result in increased WAN traffic. Caches that are placed close to the central database do not incur any WAN traffic to maintain consistency but will result in higher database access latency. We discuss the relative performance of these approaches in detail in Chapter 5.

A different approach to query caching has been followed by [Olston et al., 2005], [Sivasubramanian et al., 2006b] and [Manjhi et al., 2007]. In these systems, the edge servers do not merge different query results into a single database but stores each query result independently. We call this technique as content-blind query caching. The performance of this technique and different systems adopting this technique is discussed in detail in Chapter 3.

2.8.4. Database replication

Replication is a commonly used technique to improve database performance. Replication techniques allow us to maintain identical copies of (complete or partial) database at multiple locations in the Internet. Database replication is complex due to its consistency requirements. The level of consistency provided by the replication mechanisms and the method of maintaining that consistency largely determines the scalability of the system. In the following, we classify database replication systems based on the consistency mechanism they provide and describe some of the prominent works in this area.

Two-phase locking

The most common replication technique adopted by many database systems is to replicate the entire database and use two phase locking for consistency management (2PL) [Ullman, 1990]. In 2PL, as the name implies, the update operation is executed in two phases. In the first phase, lock is obtained from each replica (to apply the UDI query). Upon successful completion of the first phase, the UDI query is executed in the second phase. UDI queries can be submitted to any of the available replicas from where they are then synchronously applied to all other replicas. 2PL mechanism ensures that all queries are executed in the same order. Read queries are distributed between the available replicas. Systems that use 2PL to maintain consistency include MySQL Cluster[1] and Postgres-R [Kemme and Alonso, 2000].

As shown in [Kemme and Alonso, 2000], for read dominant workloads this approach will be able to achieve considerable speedup. However for write dominant workloads, it can lead to poor performance. This is due to the fact that 2PL is a fairly expensive operation. The consistency mechanism used by these systems are highly restrictive as UDI queries block execution of other incoming UDI and read queries.

Snapshot isolation

Snapshot isolation (SI) is a multi-version concurrency control mechanism used in replicated databases. Unlike 2PL, SI has a key advantage that read queries are never blocked by UDI queries. Snapshot isolation guarantees that all reads made in a transaction will see a consistent snapshot of the database. The transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot. It has been adopted by several major

[1] http://www.mysql.com

database management systems, such as Oracle[2], PostgreSQL[3] and Microsoft SQL Server[4].

Prominant middleware-based database replication systems that use SI-based consistency mechanism include [Elnikety et al., 2005], [Lin et al., 2005] and Ganymed [Plattner and Alonso, 2004]. All these system place the entire database at the replica servers and differ only in their consistency mechanisms. In Ganymed, all UDI queries are handled by a master node and all read queries are handled by the slave nodes. Upon execution of a UDI query, the writesets of the result are extracted and applied to the replicas in the same order. Read queries are applied under SI provided by the individual DBMS. While Ganymed focusses on replicating the database of a single application across a cluster of servers, DBFarm [Plattner et al., 2006] extends Ganymed to be capable of hosting database of multiple applications without loss of performance.

A potential scalability bottleneck in Ganymed is its master node. If the master node cannot handle all the write transactions, then the system cannot scale any further. This limitation is addressed by d]takemme05 who use an "update everywhere" mechanism where UDI queries can be executed at any replica and the middleware ensures the consistency of the underlying database.

In traditional replicated database systems both transaction ordering and durability (the writing of the committed database records to disk) are realized in a single action. In middleware based database replication systems, these tasks are naturally divided. The middleware determines the global ordering and the individual databases provide the durability by writing the commit record. d]tatashkent showed that this introduces a potential performance issue, since it forces some of the commit records to be written to disk serially whereas in a standalone system they could have been grouped together in a single disk write. The authors propose two different solutions to address this problem. The first solution is to move the durability from the database to the replication middleware. The second solution is to retain durability in the database and pass the global commit order from the replication middleware to the database. Their evaluations suggest that both solutions perform significant better than the basic SI-based middleware systems.

Adaptation

Database replication systems must adapt to changes in their environment. For instance, depending on the type of change (such as modifications of the load, the type of workload, the available resources, the client distribution, etc.), different

[2] http://www.oracle.com

[3] http://www.postgresql.org

[4] http://www.microsoft.com/sql/default.mspx

adjustments have to be made. In recent years, different systems have looked at the problem of adapting database-driven Web applications.

In [Soundararajan et al., 2006], the authors propose a middleware solution for database replication that dynamically allocates database replicas to applications in order to maintain application-level performance in response to either peak loads or failure conditions. The authors adopt a reactive approach where the system reacts when the average query response time falls outside an acceptable interval. In such cases, the number of database replicas is decreased/increased accordingly. Improving on their own work, in [Chen et al., 2006], the authors propose a proactive database provisioning system that increases or decreases the number of replicas based on the current system load conditions. The proposed system uses the classic K-nearest-neighbors (KNN) machine learning approach to determine the optimal number of replicas to provision for a given system condition.

2.8.5. Discussion

In this section, we presented a wide range of techniques that have been proposed in the recent years for scalable hosting of Web applications. As noted earlier, techniques such as fragment caching although widely used for serving static content may not work well for Web sites that serve highly personalized content. Edge computing systems improve the performance of compute-intensive applications as client responses are generated at edge servers located close to them. However, the centralization of the data makes it not suitable for data-intensive Web applications.

The database bottleneck of edge computing systems can be addressed by database caching or replication. As we show in the next chapter, database caching systems work well if the database query locality is high. However, current database caching solutions incur the overhead of query containment and execution which can introduce significant load under high request rates. This is one of the primary motivations for our work presented in Chapter 3.

We also surveyed numerous database replication solutions. However, most of the solutions target replication in a cluster environment where all the servers are located within a single LAN and cannot be applied directly for wide-area environments. This makes these solutions not easily applicable for edge computing environments with the edge servers located at different locations in the Internet. This is one of the key motivations behind the solutions presented in Chapters 4 and 5.

2.9. CONCLUSION

In this chapter, we have discussed the most important aspects of replica hosting system development. We have provided a generalized framework for such systems, which consists of five components: metric estimation, adaptation triggering, replica placement, consistency enforcement, and request routing. The framework has been built around an objective function, which allows to formally express the goals of replica hosting system development. For each of the framework components, we have discussed its corresponding problems, described several solutions for them, and reviewed some representative research efforts.

From this chapter, we can clearly see that the problem of scalable hosting of static Web content in a CDN environment is well understood. Works addressing the problem of building the metric estimation and client request redirection that were discussed here are applicable to CDNs hosting not only static content but also those hosting Web applications.

However, the efforts presented in the adaptation triggering, content placement algorithms and consistency mechanisms described in this chapter largely apply to systems that only host static Web objects. Replication of database-driven Web applications require replicating (some or all) its tiers [Sivasubramanian et al., 2006a]. Effective hosting of applications may warrant replication of the application code and employing appropriate database caching and replication techniques. In such cases, a major challenge is to find replication mechanisms that are not only scalable but also provide acceptable levels of consistency to the application's clients.

These issues form the central research question addressed by this thesis. We also surveyed many significant research efforts that address the problem of scalable hosting of Web applications and their shortcomings. In subsequent chapters, we propose different middleware systems that aim to improve the scalability of the database tier. Subsequently, we provide an approach by which the administrators can choose the right set of scalability techniques that needs to be applied to host their application at a desired level of performance with minimum operational costs.

CHAPTER 3

GlobeCBC: Content-Blind Query Result Caching for Web Application

In this chapter, we present the design and implementation of GlobeCBC, a database query result caching middleware for Web applications.

3.1. INTRODUCTION

As discussed in the previous chapter, CDNs employ edge computing to host Web applications (e.g., Akamai ECI [Davis et al., 2004] and ACDN [Rabinovich et al., 2003]). In these systems, the application code is replicated at all edge servers and the data are kept in a centralized server. Mere replication of code has two drawbacks: (i) each data access incurs a wide-area network latency; and (ii) the central database server (which we call the origin server) becomes a potential bottleneck. This problem has gained significant interest of the research community, resulting in a variety of middleware solutions that cache or replicate data.

As seen in Figure 3.2, data access middleware systems can be broadly classified into two types: (i) Query caching - systems that cache the results of database queries at the edge servers, and (ii) Data replication systems, which (fully or partially) replicate the underlying *database tuples*. These two approaches are suited for different kinds of Web applications. Query caching is suited for Web applications whose query workload exhibits high temporal locality and contain a small number of updates. If the workload exhibits poor temporal locality then data replication often proves beneficial [Amza et al., 2003; Cecchet, 2004; Holliday et al., 2002; Sivasubramanian et al., 2005].

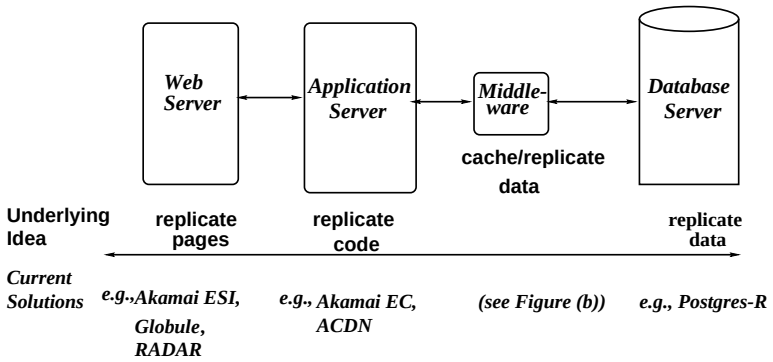


Figure 3.1: Variety of solutions that address problem of scalable Web hosting across 3 tiers

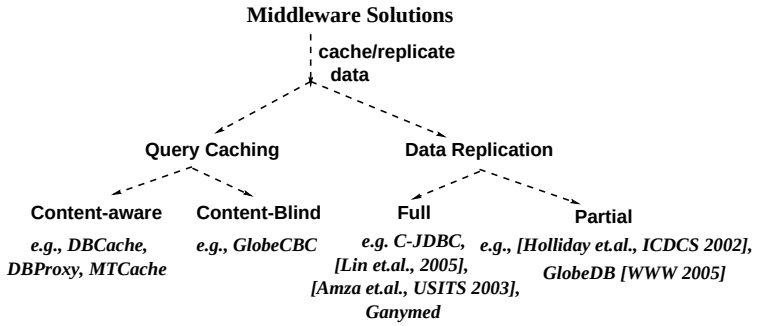


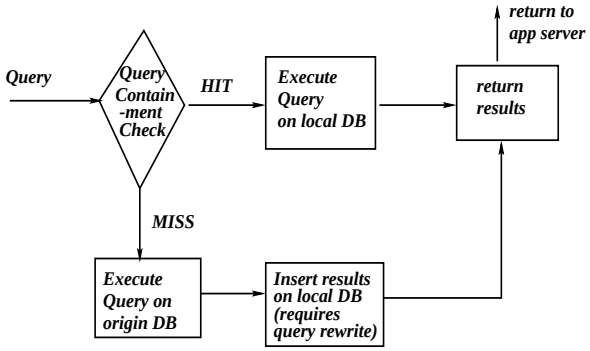
Figure 3.2: Database middleware solutions

In this chapter, we focus on the first kind of Web applications and explore the potential performance of a caching technique, which we call *content-blind* query caching, for hosting such Web applications. We present the design and implementation of *GlobeCBC*, a system that accelerates performance of Web applications by caching query results at the edge server. Unlike existing data caching middleware systems (e.g., [Bornhövd et al., 2004; Larson et al., 2004; Amiri et al., 2003a]), GlobeCBC does not merge different query results at edge server databases but stores each query result independently. At the outset, this simple approach may look very limiting. However, as we show later in the chapter, for many Web applications this approach avoids the overhead of query containment [Amiri et al., 2003b], query planning, query execution and cache management (thereby reducing the server overhead) while maintaining a high cache hit ratio (thereby avoiding wide-area network latencies). We substantiate these claims with extensive experimentations on an emulated wide-area network test-bed for different kinds of Web applications.

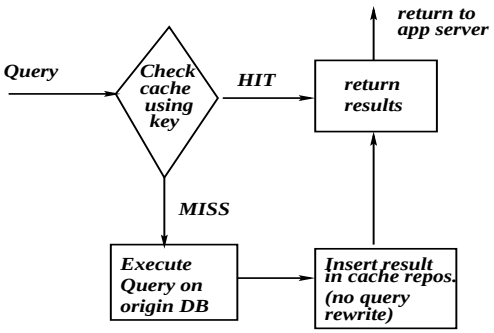
An important issue in GlobeCBC is to determine which query results to cache (or rather which item to evict from the cache), when the maximum storage capacity has been reached. This decision is governed by several factors such as the cost of each query, temporal locality, and update workload. Instead of making explicit decisions periodically, we propose the use of online cache replacement algorithms. While online cache replacement and placement algorithms are well researched in the context of Web pages, those algorithms are not always best suited to query-caching systems. We present and evaluate the performance of different query-cache replacement algorithms for GlobeCBC. We show that the best strategy is the one that takes into account both temporal locality and the execution cost of each query.

The contributions of the results presented in this chapter are twofold. First, we explore and demonstrate the potential performance benefits of content-blind query caching system for different Web applications. Second, we propose and evaluate the performance of different cache replacement algorithms and show that an algorithm that takes both the query cost and the temporal locality into account performs the best.

The rest of the chapter is organized as follows. Section 3.2 presents the design issues in building a query caching system and motivates our design choices. Section 3.3 presents the GlobeCBC architecture and Section 3.4 presents the experimentation results that compare the performance of GlobeCBC with other data caching and replication solutions. Section 3.5 presents the design and evaluation of different query replacement algorithms. Finally, Section 3.6 presents the related work and Section 3.7 concludes the chapter.



(a) Content-aware cache



(b) Content-blind cache

Figure 3.3: Cache hit and miss processing in Query Caching systems

3.2. DESIGN ISSUES

3.2.1. Data granularity

The first and foremost important design issues are to determine *what* data to cache and *how* to store them. The two possible design alternatives are *content-aware caching* and *content-blind caching*. The working of these two caching systems is pictorially described in Figure 3.3.

Content-aware caching systems run a DBMS at each edge server. As shown in Figure 3.3(a), each query received by the edge server is first checked to determine whether it can be answered with the locally cached tuples (using a *query containment* procedure). If so, the query is executed locally. Otherwise, the query

is executed on the origin server and the returned result tuples are inserted into the local database. This approach is storage efficient for queries that span a single database table as it does not store redundant tuples. Results of queries spanning multiple tables are usually stored separately.

Here, we pursue the relatively less explored content-blind caching approach (notably, a similar approach was explored [Olston et al., 2005]; we discuss it in detail in Section 3.6). The key difference between content-aware and content-blind caching is that the latter does not merge different query results and stores each result separately. Each query can be answered from the cache only if the result of the same query has been cached. There are several advantages to this approach. First, the process of checking if a query is cached or not becomes trivial and scales very well even for high loads (whereas query containment approaches are relatively expensive). Second, by caching results directly, the edge servers avoid the overhead of database query planning and query execution. In fact, edge servers do not even need to run a DBMS at all. This is especially beneficial under high load. Finally, cache replacement is quite simple as each result is stored independently.

The content-blind caching approach also has some shortcomings. First, by storing query results independently it possibly stores redundant data. This drawback also exists to a lesser extent in content-aware caching systems, as they also need to store the results of queries spanning multiple tables independently. Second, by skipping the query containment procedures, certain queries may not not be answered locally even if all the required tuples are available locally. However, as we show later in our experiments, this approach performs better than its caching counterparts for different Web applications.

3.2.2. Cache control and placement

Other design issues that need to be addressed are to decide who selects which queries to cache (*cache control*) and how the selection is made (*cache placement*). Cache control can be *centralized*, where a central server collects query access patterns and decides which queries must be cached in which edge server or *distributed*, where each edge server independently decides which items to cache (or to evict from the cache). We choose the second option because it scales naturally with the addition of new edge servers and reduces the control overhead in the origin server.

Cache placement decides on which set of query results to cache. This is important if the edge server does not have enough resources to cache all query results. Moreover, the problem of determining which results to keep in the cache gains significance, if the cached results are always kept in main memory instead of disk. Cache placement has direct impact on the cache hit ratio and thus also on client latency and throughput, as well as the network traffic between the edge server(s) and

the origin server. We choose an online caching algorithm by which an edge server caches the results of all queries it receives (unless explicitly specified). When the maximum storage capacity is reached, each edge server will run a cache replacement algorithm to determine the least beneficial cached query to evict from the cache. Although cache replacement algorithms are well researched in the context of static Web pages, only few efforts have been conducted to explore them in the context of database query caching [Dar et al., 1996; Amiri et al., 2002].

3.2.3. Consistency

Cached query results need to be updated or invalidated when the database is updated. As seen in the previous chapter, the problem of maintaining cache consistency has been extensively studied in the context of static Web pages. However, consistency maintenance in query result caching is different from Web page caching for two reasons. First, an update to a single database record can affect multiple query results while at the same time it is more difficult to determine which cached result must be invalidated. In contrast, an update to a Web page usually affects only a single object. Second, the ratio of number of updates to number of reads is much higher when dealing with database-driven Web applications.

These two issues make the problem of maintaining consistency in a query-caching system challenging. To address them, we assume that the query workload consists of a fixed set of read and write query templates. A query template is a parameterized SQL query whose parameter values are passed to the system at runtime. This scheme is deployed, for example, using Java's prepared statement. Examples of parameterized read query templates include $QT1$: "SELECT price, stock, details FROM book WHERE id=?" and $QT2$: "SELECT price, stock, details FROM book, author WHERE book.name LIKE (?) AND author.name = ?". The following is an example of update template $UT1$: "UPDATE price=price+1 FROM book WHERE id=?".

In our system, we expect the developer to specify a priori which query template conflicts with which update template. Based on this, whenever an origin server receives an update query, it invalidates all results belonging to the conflicting templates. Although this assumption of data access through pre-defined query templates curtails the flexibility of the system in being able to handle new query types, it suits Web applications well.

Template-based invalidation prevents from invalidating all cached results upon each database update. However, there can still be a problem if the number of updates to the database is high or if there exists a significant number of conflicts between the query templates and update templates. In such cases, massive invalidations will lead to a low cache hit ratio, increased network traffic and increased server load, thereby leading to high client latencies. Typically, this problem is ad-

dressed by adoption of *weak consistency*. In this chapter, we explore how different forms of weak consistency can be easily integrated into our system and show the potential scalability benefits obtained by them. We explain the consistency properties of our system in detail in the next section.

3.3. SYSTEM ARCHITECTURE

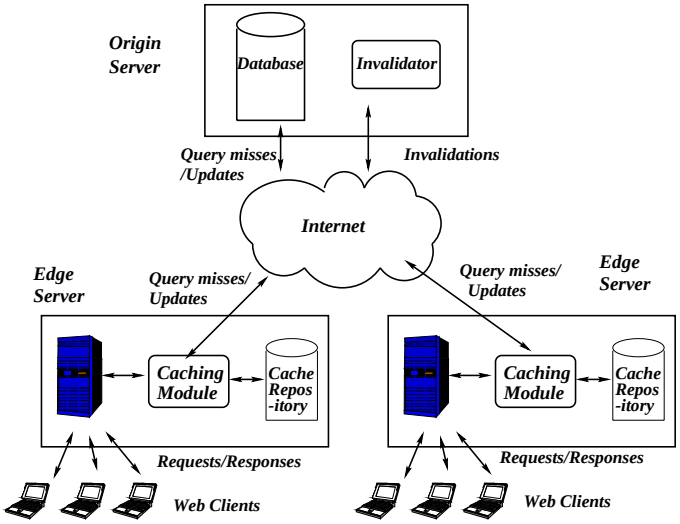


Figure 3.4: GlobeCBC: System Architecture

The architecture of edge computing infrastructure running GlobeCBC is presented in Figure 3.4. An application is hosted by edge servers located across different regions in the Internet. Communication between edge servers usually traverses the wide-area network incurring wide-area latency. Each client is assumed to be redirected to its closest edge server using enhanced DNS-based redirection [Dilley et al., 2002; Fei et al., 1998]. For each session, a client is assumed to be served by only one edge server. We assume that the application code is replicated at all edge servers. Furthermore, for each application, one of the edge servers acts as the *origin server*. The origin server hosts the complete database of the application.

The two key components of our system are the *caching module* and the *invalidator*. The caching module is the middleware that runs in each edge server and intercepts all query execution requests between the application code and the origin server database. It is responsible for (i) determining if the requested read query's

result is available in the local cache repository, (ii) returning query results (either from the local cache repository or by executing the query at the origin server) and (iii) adding/replacing new query results into the cache.

The invalidator is a stand-alone service that runs at the origin server. It is responsible for monitoring all update queries addressed to the database server, determining the list of query templates to invalidate and issuing invalidations. We will discuss the design and implementation of these components below.

3.3.1. Caching module

The caching module is implemented as a PHP driver and can be added as a module to the Apache Web Server[1]. As said earlier, we assume that the application's query workload consists of a fixed set of read and write templates. Each query is identified by the module using the structure: $< template\text{-}id, query\ parameters >$ as key.

When a read query is issued, the caching module checks if the query result is cached in the local repository (using its key as the unique identifier). If found, then the result is returned immediately. Note that this is different from traditional edge database caching systems as the cached units are not the result tuples but the result structures. This allows the caching module to return the query result immediately without query planning and execution overheads.

If the incoming query is not found in the cache, then the query is executed at the origin server. Upon successful execution, the result is stored in the local repository. Note that the local cache repository does not use a DBMS to store query results. Instead, in our system, we store the result of each query as a separate file in the local file system. Of course, to improve performance it can also be kept in main memory. The caching module subscribes at the invalidator to the invalidation channels corresponding to the cached template-*id* (if it had not subscribed already). This allows the edge server to be informed of the updates to the underlying database that can *possibly* affect the cached results. This process of invalidation is described in detail in the next section.

When the caching module receives a write query (i.e., update/insert/delete), it is executed on the origin server's database. Subsequently, it sends a message (asynchronously) to the invalidator with the template identifier and the parameter (if applicable) of the write query. Note that since invalidation messages are sent asynchronously to the edge servers, the system does not provide strong consistency. We assume that this is acceptable for the hosted Web application. However, if a particular query template can not tolerate any staleness in its result, then the application developer can disable caching for the concerned query template.

[1] http://www.apache.org/

3.3.2. Invalidator

The invalidator is a stand-alone service that runs in the origin server and is responsible for invalidating the cached results at edge servers when they become stale due to updates. As noted earlier, we assume that the Web application's workload consists of a fixed set of read and write query templates. Furthermore, we assume that the conflicts between read and write templates are marked offhand either manually by the developer or using automated code parsers. For example, a query template whose instances add a new book to the book table (e.g., "INSERT into books VALUES(..)") conflicts with query instances that find the newly added books.

The invalidator maintains such a conflict map for all the write query templates. For each write template, the invalidator maintains a publish-subscribe channel. Edge servers that cache a query result of template instance *id* subscribe to the channels of write template(s) that *id* conflicts with.

When a write query is performed on the origin server's database, the invalidator receives a message from the edge server's caching module. Upon its receipt, the invalidator sends the write query key (and the query parameter, if applicable) to all the subscribers in the channel corresponding to the write identifier. Upon receipt of the invalidation message, the caching module in each edge server invalidates instances of the query templates that conflicts with the write template. We discuss the process of template-based invalidation in detail in the next section.

3.3.3. Fine-grained invalidation

Template-based invalidation helps in reducing the number of invalidations performed, provided there is no conflict between a result and update query even though they operate on the same database table(s). For example, in a bookstore application, the cached result of a query to find the best selling books is not affected by an update query to the book table for changing the price of a book. However, the cached result should be invalidated if a new book order is placed.

Such a simple coarse-grained template-based invalidation is sometimes too conservative as it invalidates all instances of a conflicting query template. For example, consider the following template: QT1: "SELECT price, stock, details from book where id=?" and update query template UT1: "Update price=price+1 from books where id=?". Using a simple coarse-grained invalidation scheme, an update to even a single book (e.g., $< UT1, 100 >$) will invalidate all cached instances of QT1.

To avoid this conservative invalidation, a simple extension is to take into account the parameter of the updated item and invalidate only those cached queries that are affected by the updated item. However, the system can determine which

cached queries are affected by an update, without examining the content, only for simple queries (i.e., SQL queries which access information based on the primary key). So, in the above example, when the system receives an update query with key: <UT1,100 >, the invalidator invalidates only the cached item whose key is <QT1, 100>, provided QT1 is a simple query. As shown later, this simple extension improves the performance of applications whose workload has lots of simple queries, such as the TPC-W benchmark.

3.3.4. Tunable consistency

Template-based invalidations help in reducing the number of invalidations to an extent. However, for Web applications that have a large number of read-write conflicts, the system will generate large numbers of invalidations. This can cause a poor cache hit ratio, leading to increased wide-area network traffic, increased origin server load, and therefore increased client latency. Applications that have these characteristics can usually scale only by employing weak consistency.

As discussed in the previous chapter, weak consistency protocols are usually employed along one of the following axes: *time*, *order of operations* or *value*. In our system, we provide interfaces to the application designers to *tune* their consistency bounds based on *time* and *order*. We do not support value-based mechanisms as our caching module is content-blind.

For time-based mechanisms, the system expects the application developer (or the system administrator) to set TTI_i (time-to-invalidate) values for each update template, U_i. Subsequently, if an update query of type U_i is received by the invalidator, the invalidator starts a timer for TTI_i seconds (unless a timer has already been started). After TTI_i seconds, all queued invalidation messages for conflicting read templates are sent.

For order-based weak consistency mechanisms, we employ a mechanism similar to N-ignorant transactions. The system requires the application developer (or the administrator) to set the bound for each update template U_i, regarding the number of updates the invalidator can tolerate before invalidating the conflicting read templates (Max_Upds_i). To implement this mechanism, the invalidator maintains a counter that keeps track of the number of updates it has received for each update template (Num_Upds_i). When $Num\_Upds_i \geq Max\_Upds_i$, the invalidator sends out the invalidation messages and resets Num_Upds_i to 0.

The system can also support a combination of both time and order-based weak consistency mechanisms. Application developers or system administrators who want to get higher scalability using weak consistency, can simply tune one (or both) of these two parameters, TTI and Max_Upds. We call this means of employing weak consistency as *tunable consistency*.

3.4. PERFORMANCE EVALUATION

In this section, we compare the performance of content-blind caching to other solutions. We consider two different applications, a news Web site modelling http://slashdot.org and the TPC-W benchmark, an industry standard e-commerce benchmark that models an online bookstore. We chose these two applications for their different data access characteristics. For example, in a typical news forum, most users are usually interested in the latest news and so the workload will usually exhibit high locality. On the other hand, in a bookstore application, the shopping interests of customers can be different thereby leading to much lower query locality. This allows us to study the behavior of content-blind caching for different data access patterns.

3.4.1. Performance results: Slashdot application

In this section, we present results of experiments performed using the RUBBoS benchmark, an open source benchmark that models an online news forum application similar to slashdot.org[2]. The benchmark is written in PHP. Its database consists of five tables, storing information regarding users, stories, comments, submissions and moderator activities. The database is filled with $500,000$ users, out of which 10% have moderator privileges, and $200,000$ comments. The size of the database is approximately 1.5 *GB*.

We deployed the GlobeCBC prototype across 3 identical edge servers each with dual-processor Pentium III 900 Mhz CPU, 1 *GB* of memory and a 120 *GB* IDE hard disk. Each edge server uses an Apache 2.0.49 Web server with PHP 4.3.6. We use PostgreSQL 7.3.4 for our database servers and PgPool for pooling database connections[3]. The origin server uses an identical configuration as the edge servers except that it acts just as a backend database and does not run a Web server. We emulate a wide-area network (WAN) among the servers by directing all the traffic to an intermediate router which uses the NISTNet network emulator[4]. This router delays packets sent between the different servers to simulate a realistic wide-area network. In the remaining discussion, we refer to links via NISTNet with a bandwidth of 50*Mbps* and a latency of 100ms as WAN links, and 100*Mbps* and 0 latency as LAN links. Note that these bandwidth and latency values are considerably optimistic, as the Internet bandwidth usually varies a lot and is constantly affected by network congestion. These values are chosen to model the best network conditions for a CDN built on an Internet backbone and are the *least favorable* conditions to show the best performance of any data caching or

[2] http://jmob.objectweb.org/rubbos.html

[3] http://pgfoundry.org/projects/pgpool/

[4] http://snad.ncsl.nist.gov/itg/nistnet/

replication system. Any lower bandwidth or higher latency will only boost the performance of caching systems. For example, if the WAN delay in the route is set to a higher latency, say 500ms, then for edge computing infrastructures that do not replicate data, each database query will incur at least 500ms round trip latency (and more if each query is transmitted in multiple TCP packets). This value will therefore boost the performance of data caching solutions as they can answer queries locally.

We use three client machines to generate requests addressed to the three edge servers. The client workload for the system is generated by Emulated Browsers (EBs). The run-time behavior of an EB models a single active client session. Starting from the home page of the site, each EB uses a Customer Behavior Graph Model (a Markov chain with various interactions with the Web pages in a Web site as nodes and transition probabilities as edges) to navigate among Web pages, performing a sequence of Web interactions [Menasce, 2002]. The behavior model also incorporates a think time parameter that controls the amount of time an EB waits between receiving a response and issuing the next request, thereby modelling a human user more accurately. The user workload contains more than 15% interactions that lead to updates.

We evaluated four system architectures using the aforementioned setup: (i) *Edge Computing:* the code is replicated in every edge server while the data remain centralized at the origin server. (ii) *Full replication:* the code and database are fully replicated in the edge server. All updates are performed at the origin server and are propagated asynchronously to the edge servers. (iii) *Content-aware caching:* the edge servers run a Web application server and a DBMS. The key difference between this system and full replication is that the edge servers run the query locally only if the *query containment* check results in a hit. We implemented the algorithm described in [Amiri et al., 2003a] for query containment. (iv) *Content-blind caching:* The edge servers run a Web application server and a cache repository. The origin server runs the database. Unless stated otherwise, in content-blind caching schemes, the invalidator is configured with the strongest possible consistency, $TTI = 0$ and $Max\_Upds = 0$. Note that full replication, content-aware and content-blind caching systems provide the same level of consistency while only the edge computing architecture provides the stronger level of consistency.

All experiments are started with a cold cache. The system is warmed up for 20 minutes, after which the measurements are taken for a period of 90 minutes. We assume the edge server to have infinite cache capacity, i.e., the size of cache repository was not restricted because we did not want the effect of cache replacement algorithms to affect the performance of content-aware and content-blind caching systems. We study the performance of different cache replacement algorithms in

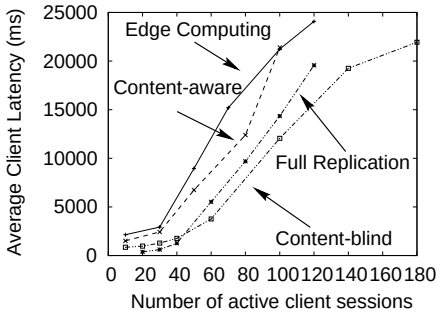


Figure 3.5: Effect of Workload on Slashdot application

the next section. In all experiments, we measure the end-to-end *client latency*, which is the sum of the *network latency* (the time spent by the request traversing the WAN) and *internal latency* (the time spent by the request in generating the query responses and composing the subsequent HTML pages).

Results for WAN experiments

We first evaluated the client latency for different architectures for only one edge server and the origin server. We studied the client latency for different client workloads. The results of our experiment are shown in Figure 3.5.

As seen in the figure, content-blind caching performs the best in terms of client latency (except for low loads) while edge computing performs the worst in all cases. It can also be seen that content-blind caching sustains higher load than full replication and the content-aware caching. This is quite remarkable considering that content-blind caching provides the same level of consistency. The edge computing infrastructure performs worse than the other architectures. This is due to the fact that all data accesses incur a WAN latency in addition to the origin server becoming the scalability bottleneck thereby increasing the internal latency in generating query responses.

Full replication system performs marginally better than content-blind under very low client loads (up until 30 client sessions). This is because in a full replication system each query is answered locally thereby avoiding any wide-area network latency. During low workloads, the internal latency incurred in generating a query response is lower than the network latency incurred in answering a query. Therefore, full replication system performs marginally better in such workloads. On the other hand, content-blind caching outperforms full replication during high workloads for two reasons. First, the application's workload exhibits good temporal locality thereby avoiding WAN latency. Second, the internal latency in generating a query response, for a content-blind caching system, is much lower than

that of full replication as the caching system avoids query planning and execution latency.

Content-blind caching performs better than content-aware caching for all client workloads. Even though both systems are equally capable of capturing temporal locality, the former incurs more latency in generating query responses as it incurs the overhead of query containment, cache management (inserting and invalidating caches), query planning, and execution. On the other hand, content-blind caching avoids these overheads by storing the results of query instances separately, thereby resulting in lower client latency. Furthermore, the internal latency overhead increases with increase in client load, as we discuss next.

Understanding internal latencies

To better understand the effect of client workloads on query execution overheads and subsequently on the internal page generation latency, we isolated a single edge server and studied the latency in generating a page for two different systems: (i) a Pentium III dual processor machine that runs a vanilla Apache server and PostgreSQL 7.3.4 (WAS-DB) and (ii) the same setup that runs the content-blind caching module, in addition to the Web server and database server (WAS-Cache-DB). Also, we evaluated the client latency for read-only client sessions for different loads. In both setups, the network latency incurred for generating a page is set to zero as both the Web server and database reside on the same machine. The objective of this study is to measure the potential gain in internal latency by the use of content-blind caching which avoids query planning and execution latencies.

The results of our study are given in Figure 3.6. As seen in the figure, the internal latency of a system that employs content-blind caching (WAS-Cache-DB) is much lower than that of the traditional WAS-DB. In particular, the difference grows up to an order of magnitude for high loads. This study thus gives a clear insight into the potential gains of content-blind caching. It also explains the better performance of content-blind caching in comparison to full replication and content-aware caching system, which always incur the query planning and execution overheads.

Study with multiple edge servers

As a next experiment, we compared the average client latency of a content-blind caching system with a single edge server to one with three edge servers. For the latter case, the client load was equally divided among the three edge servers. As seen in Figure 3.7, the system with three edge servers performs significantly better in terms of client latency (by a factor of 2) compared to the single edge server system. This can of course be easily understood as the edge servers share

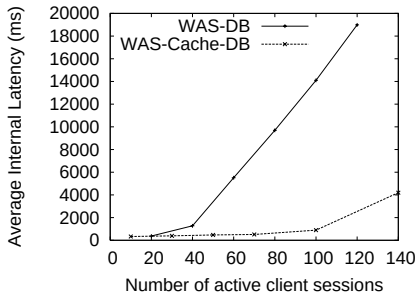


Figure 3.6: Comparison of internal latency for plain 3-tier architecture with and without content-blind caching system

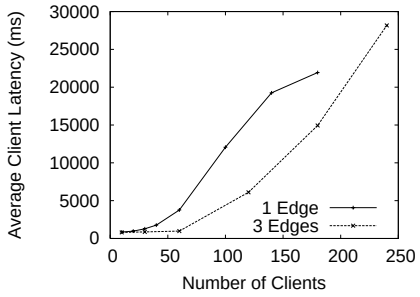
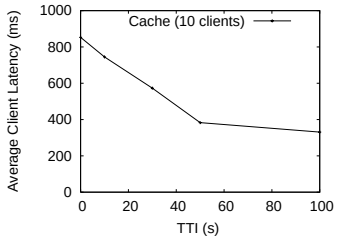


Figure 3.7: Comparison of client latency for single edge server and 3 edge server system

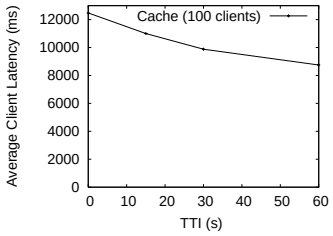
their client load. The difference in client latency is negligible during low loads as a single edge server is able to sustain the client load under these conditions. Under high loads, the 3-edge server system is able to sustain more load. However, as can be seen from the figure, the system does not offer a linear improvement and can only sustain 50% more load than the single edge server. This is because in both cases the database in the origin server remains the central scalability bottleneck.

Tunable consistency

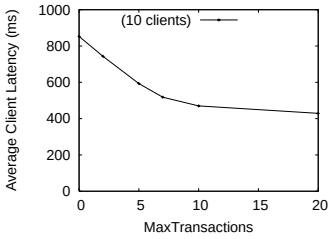
For high workloads with a large number of updates (recall that the fraction of update interactions is around 15%), immediate invalidations increase the system load leading to a higher client latency. If more scalability is desired, such situations demand relaxed consistency models. As noted earlier, we adopt a tunable consistency model in which application developers or system administrators specify the limits of weak consistency along the axes of time (TTI) and order (Max_Upds).



(a) Effect of TTI for 10 clients



(b) Effect of TTI for 100 clients



(c) Effect of *Max_Upds* for 10 clients

Figure 3.8: Effect of TTI and *Max_Upds*

In this experiment, we study the gains in client latency of relaxing consistency with different values of TTI and *Max_Upds* in a system with a single edge server connected to the origin server through a WAN link. In the first set of experiments, we fixed *Max_Upds* at 0 and study the impact of TTI on client latency for a load of 10 and 100 client sessions. The results are given in Figure 3.8(a) and 3.8(b). As seen in these figures, for a TTI of 1 minute, we gain a factor of 3 in client latency under low load and 30% improvement under high load. The difference in gains of client latency is due to the increase in internal latency.

In the second set of experiments, we studied the effect of *Max_Upds* on client latency. The results are given in Figure 3.8(c). With a *Max_Upds* value of 10, we can now gain a factor of 2 in client latency.

The objective of this study is not to recommend the best value of TTI or *Max_Upds*, as the best values depend on the application needs and system costs. This experiment is rather meant to demonstrate that, provided the application can support it, controlled relaxation of consistency can produce significant gains in performance.

3.4.2. Performance results: TPC-W benchmark

We evaluated the performance of different systems for the TPC-W benchmark, an industry standard e-commerce benchmark that models an online bookstore like Amazon.com [Menasce, 2002]. We used the open source PHP implementation of TPC-W.[5] We modified the client workload behavior such that the book popularity follows a Zipf distribution (with $alpha = 1$), which was found in a study that observed data characteristics of the Amazon online bookstore [Brynjolfsson et al., 2003].

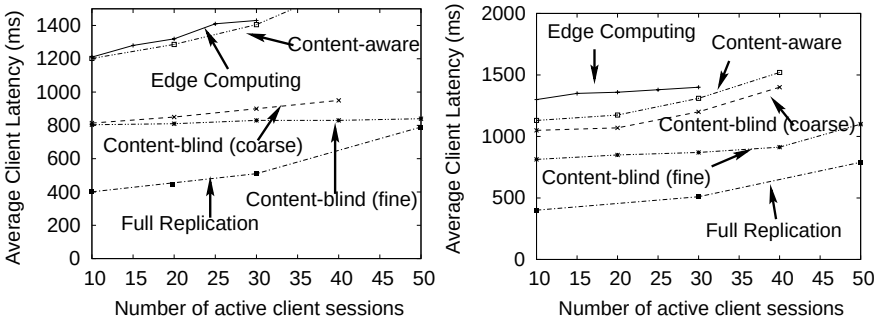
For this application, we studied the performance of four systems: edge computing, content-aware caching, content-blind caching and full replication. For content-blind caching, we studied the client latency for two kinds of invalidations: coarse-grained and fine-grained invalidation. The objective is to examine the potential benefits in employing the finer invalidation. Recall that fine-grained invalidation takes into the account the parameter of the update template to invalidate conflicting simple queries.

We studied the client latencies of these systems for two kinds of workloads: browsing (which consists of 95% browsing and 5% ordering interactions) and ordering workload (50% browsing and 50% shopping interactions). As seen in Figure 3.9, edge computing performs the worst in client latency while full replication performs the best. In particular, full replication performs better than content-blind caching because the TPC-W benchmark workload exhibits poor temporal locality and yields a hit ratio of at most 35% in our experiments. In such cases, data replication helps as each query can be answered locally. However, during high loads, the gain in latency decreases as the internal latency of full replication increases. These results are in line with our earlier results, where we demonstrated that applications whose workload exhibit poor temporal locality or has many updates are best hosted using data replication schemes [Sivasubramanian et al., 2005].

Content-blind caching performs better than content-aware caching as both are able to capture temporal locality and the former benefits from reduced internal latency. Note that the gain in client latency in content-blind caching compared to edge computing in TPC-W benchmark is only about 50%, while it was a factor of 3 for the Slashdot application. This is again due to the poor temporal locality exhibited by the workload.

Between the two invalidation mechanisms of content-blind caching system, the fine-grained one consistently performs better than its coarse-grained counterpart. The difference in latency is especially high for the ordering workload. This is because the fraction of simple queries in this workload is higher than that of the browsing workload (recall that simple queries are those accessing data in a single

[5] http://pgfoundry.org/projects/tpc-w-php/



(a) Browsing **(b)** Ordering

Figure 3.9: Performance of our architecture compared to edge computing for TPC-W benchmark

table with primary keys). As fine-grained invalidation performs fewer invalidations for instances of simple query templates, it improves overall client latency[6].

3.4.3. Discussion

As can be seen with these experiments, content-blind caching performs better than the other architectures while providing the same level of consistency guarantees, provided the application's data workload exhibits high temporal locality. Furthermore, for these applications, we also showed that our system allows administrators to achieve higher scalability employing relaxed consistency by simply tuning two variables, TTI and Max_Upds.

On the other hand, for applications that exhibit poor locality (such as the TPC-W benchmark), data replication schemes perform better than content-blind caching. Our hypothesis is that there exists no single solution that can perform the best for all Web applications and workloads. Different techniques are optimal for different applications. As we show in Chapters 4 and 5, applications with poor temporal locality can be hosted scalably using data replication. As shown in this section, applications that exhibit high locality can benefit to a large extent from content-blind caching and outperform its replication counterparts. The performance of an integrated system that employs a combination of GlobeCBC and database replication is presented in Chapter 5.

[6]The performance difference between these invalidation schemes in Slashdot application was negligible. This is because fine-grained invalidation improves performance only in the presence of simple queries. In the Slashdot application, the majority of the query workload consists of complex queries.

3.5. CACHE REPLACEMENT

An important issue in any caching system is to determine which query results to cache and which ones to evict from the cache. This is usually not an issue if the edge server has unlimited memory and storage resources. However, in CDNs, the edge servers are usually simple desktop servers with limited memory resources and simple disk access resources (for example, most of the servers use IDE disks). In such an environment, it is desirable to keep as many query results in main memory as possible and hence the problem of which results to keep in the cache gains significance. This issue is especially relevant in a collaborative CDN environment such as Globule [Pierre and van Steen, 2006] where the edge servers are usually end-user machines.

3.5.1. Cache replacement

Cache replacement is simple in content-blind caching compared to content-aware caching. For example, in the latter, evicting a query result from the cache requires appropriate checks to ensure that query containment conditions of other query templates are not violated as results are merged. However, since content-blind caching stores results independently, each result can be replaced independently. GlobeCBC uses an online cache replacement mechanism to determine which results are more beneficial to retain in the cache when the cache is full. The query result replacement mechanism works as follows: When the cache is full, it must select one or more cached results to be removed so that resource constraints are met again.

The performance of the caching system depends to a large extent on the effectiveness of the cache replacement algorithm. An ideal cache replacement algorithm must take into account several metrics such as temporal locality, cost of the query, update characteristics of the database, etc. In this chapter, we evaluate the average query latency of different cache replacement algorithms that operate on one or more of these metrics to find the one most suited for our system.

We designed and evaluated the performance of the following cache replacement algorithms:

1. *Least Recently Used (LRU)*. LRU always deletes the recently used cached result with the new result. The intuition is that the most recently accessed queries are most likely to be accessed again. Previous research on Web page caching suggests that strategy performs best if all cached items are equal.

2. *Least Cost (LC)*. Each query takes different time to execute in the database server. This is usually modelled by a *query cost* parameter, which is used in

the query planning. LC replaces the new result with the cache item that has the least query cost. The intuition is that a cache hit on a high cost query is more valuable than hit on a low cost query as it will offload the origin server to a great extent.

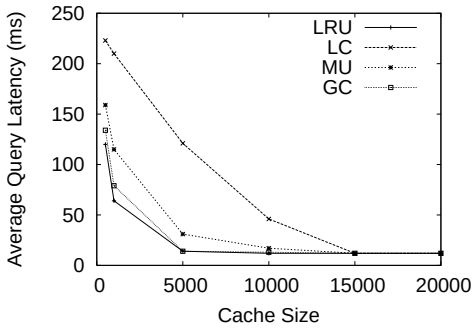
3. *Most Updated (MU)*. Each edge server counts the number of invalidations received by each template. The system replaces the result whose parent template is most invalidated. The intuition is caching the least updated queries might improve the cache hit ratio.

4. *Greedy-Cost (GC)*. LC optimizes on internal latency but does not exploit the locality of requests. On the other hand, LRU exploits the locality of requests but ignores the individual characteristics of the cached items. Greedy-cost aims to capture the best properties of these two algorithms. The algorithm associates a value, C_i, with each cached query result q_i. Initially, when a result is brought into the cache, C_i is set to be the time incurred in bringing it. When a replacement needs to be made, the result with the lowest C value, C_{min}, is replaced, and then all results reduce their C values by C_{min}. If a query result is accessed, its C value is restored to its initial value. Thus, the C values of recently accessed results retain a larger portion of the original cost than those of results that have not been accessed for a long time. This way, GC exploits temporal locality and takes query cost into account at the same time. This algorithm is similar to the Greedy-Dual algorithm used in Web caching [Cao and Irani, 1997].

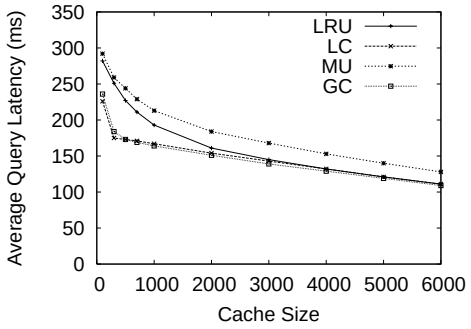
3.5.2. Evaluation results

We evaluated the performance of different cache replacement algorithms using trace-driven simulations. We collected query traces of the RUBBoS and TPC-W benchmark (shopping mix). Each trace represents two hours of execution with a workload of 10 active client sessions. The cost of each query template is assigned as the average query response time of all query instances of the given template. This was also used as the internal latency in calculating the overall query response time. We fixed the WAN latency to be 100ms.

We replayed these traces with different replacement algorithms, assuming that the latency of a cache hit is 0 and the latency of a cache miss is the sum of internal latency and WAN latency (if incurred). Note that the average query latency is different from the client-perceived latency as each client request may trigger multiple query requests to generate a page. Our experiments were started with a cold cache and the measurements were taken for the entire run. We report the average query latency for various cache sizes.



(a) Slashdot



(b) TPC Shopping

Figure 3.10: Performance of different cache replacement algorithms for different cache sizes

The results of our experiment are given in Figure 3.10. For both applications, GC performs the best or close to the best. LRU performs the best for the Slashdot application due to the high temporal locality of the query workload and GC performs almost as good as LRU. However, LRU performs poorly for the TPC-W application as the query workload does not exhibit high locality. As a consequence, it constantly replaces high-cost queries (which form almost the 30% of query workload). This leads to an increase in internal latency of queries.

LC performs poorly for the Slashdot application as it is not capable of exploiting temporal locality. However, it does well for TPC-W as it retains the high cost queries and hence is able to optimize on high cost queries which exhibit high locality. MU performs poorly for both applications as it does not exploit locality and also does not take into account the cost of the queries. MU can be useful only if the workload generates a significant number of invalidations. However, in the TPC-W application, even though many updates are issued, the template-based query invalidation does a reasonable job in reducing the number of invalidations thereby reducing MU's performance gain.

In summary, among all strategies, GC performs (close to) the best by virtue of being able to exploit locality and keeping a good balance in managing high-cost queries.

3.6. RELATED WORK

As shown in Figure 3.1 and described in the previous chapter, a number of systems have been developed to handle Web application hosting. As we have shown in our evaluations, edge computing systems perform poorly as they incur WAN latency for each data access and direct all the load to the central database.

Database caching systems such as [Amiri et al., 2003a], [Amza et al., 2005], and [Luo and Naughton, 2001] aim to address this limitation. These systems fall in the category of *content-aware* caching systems. They store the database tuples that form the results of queries in the DBMS running in the edge server (provided the query is on a single table) and merge different query results. A similar approach known as semantic caching was also proposed for client-server database systems in [Dar et al., 1996]. These systems are built to be very flexible and can support different types of applications (i.e., not just to template-based Web applications).

However, this flexibility comes at a cost. As shown in Figure 3.3(a), in these systems, each query needs to be subject to a local query containment check procedure to determine if the edge database server has all the data required to answer the query completely. Even if the containment test results in a cache hit, the system incurs query planning and execution overhead. Moreover, inserting and remov-

ing items from the local cache also is a non-trivial process. On the other hand, GlobeCBC is limited in flexibility and is suited only for Web applications whose workload usually consists of a small set of read and write templates. This allows the system to avoid query execution overheads and achieve better client performance even under heavy loads, provided the workload exhibits good locality as we have shown in our performance results. Furthermore, content-blind caching also allows the system to add or remove items from the cache easily as each item is treated and stored separately.

Many middleware systems have been proposed for scalable replication of a database in a cluster of servers [Amza et al., 2003; Cecchet, 2004; Chen et al., 2006; Lin et al., 2005; Plattner and Alonso, 2004]. However, the focus of these works is to improve the throughput of the backend database within a cluster environment. On the other hand, the focus of our work is to improve the client-perceived latency in a CDN environment where edge servers are spread across a wide-area network. Furthermore, as we showed earlier, data replication is beneficial if the workload exhibits poor locality and low number of updates. If the number of updates increases, then autonomic replication solutions can be envisaged [Sivasubramanian et al., 2005].

In [Olston et al., 2005], the authors investigate the use of a similar query caching system. Compared to their work, we explore the potential performance gains of content-blind caching systems for different applications through extensive evaluations of different edge computing architectures in an emulated wide-area network. We also propose and evaluate different cache replacement algorithms to address the case of resource-constrained edge servers. These issues and experiments were not studied in [Olston et al., 2005].

In [Manjhi et al., 2007], the authors study the issue of privacy and security if the database queries are cached in untrusted edge servers. In such environments, the issue of ensuring the privacy and security of the cached data is an important requirement. To this end, the authors propose to encrypt the query results and invalidations. However, this makes the problem of invalidation harder as the caching modules in edge servers cannot inspect the encrypted queries to determine which queries need to invalidated. To address this, the authors propose a technique called invalidation clues that enables caches to reveal little data to the third party servers yet limit the number of unnecessary invalidations. Compared to this work, we focus on an enterprise CDN environment where all edge servers are assumed to be trusted. Hence, we do not face this problem. However, the techniques presented in [Manjhi et al., 2007] can be easily applied to GlobeCBC as well.

3.7. CONCLUSION

In this chapter, we presented GlobeCBC, a content-blind query caching middleware for hosting Web applications in an edge computing infrastructure. Unlike existing data caching middleware systems, content-blind caching systems do not merge different query results and store the query results as result structures independently. We studied the potential performance of this approach using extensive experimentations on our prototype implementation and compared it with the other systems over an emulated wide-area network. Our evaluations show that content-blind caching performs well in terms of client latency for applications that exhibit high locality and is also able to sustain more load by offloading the origin server database. We also showed that when applications can support it, the system administrators can still improve performance by relaxing the data consistency in a simple fashion. We proposed and evaluated different online query replacement algorithms which will be useful for resource-constrained edge servers. In our evaluations, we found that the best algorithm must take into account both the query execution cost and the temporal locality.

Even though GlobeCBC performs very well for applications with high query locality, it is outperformed by database tuple replication when the workload has poor locality. This shortcoming is addressed by the database replication techniques proposed in the subsequent chapters. As shown in Chapter 5, integrating GlobeCBC with the our database replication solutions allows the database tier to achieve significantly higher performance and scalability compared to existing database replication solutions.

CHAPTER 4

GlobeDB: Autonomic Replication for Web Applications

4.1. INTRODUCTION

In this chapter, we explore a different approach to hosting Web applications. The idea is to replicate not the pages (as done in fragment caching [Challenger et al., 2005; Datta et al., 2002; Douglis et al., 1997; Li et al., 2003; Rabinovich et al., 1997]) but the data residing in the underlying database. This way, the dynamic pages can be generated by the edge servers without having to forward the database requests to a centralized server.

As observed in the previous chapter, database replication performs better than query result caching if the temporal locality of database queries is low. Moreover, in cases where the temporal locality of the database queries and their update rate are high, an application may benefit by moving (part of) the underlying database, along with the code that accesses the data, to the location where most updates are initiated. This allows the edge servers to generate documents locally thereby resulting in improved client-perceived performance. However, replication of the entire application data at all edge servers can result in significant network usage due to the update traffic incurred in keeping the replicas consistent.

We are thus confronted with the problem of sustaining performance through distribution and replication of (parts of) an application, depending on access and usage patterns. In general, we need to address the following three issues: (1) determine which parts of an application and its data need to be replicated, (2) find where these parts need to be placed in a wide-area network, and (3) decide how replicated data should be kept consistent in the presence of updates.

These issues are currently handled by human experts who manually partition the databases of a Web application, and subsequently distribute the data and code

among various servers around the Internet. Not only is this a difficult and time-consuming process, it is also not very feasible when usage and access patterns change over time [Sivasubramanian et al., 2003]. In addition, consistency in these systems tends to be ad-hoc and largely application-dependent.

In this chapter we present the design and implementation of GlobeDB, a system for hosting Web applications. GlobeDB handles distribution and partial replication of application data *automatically* and *efficiently*. It provides Web-based data-intensive applications the same advantages that CDNs offer to traditional Web sites: low latency and reduced network usage. We substantiate these claims through extensive experimentation of a prototype implementation running the TPC-W benchmark over an emulated wide-area network.

Using GlobeDB, we demonstrate that the configuration of Web applications for data and code replication can be largely automated, and in such a way that it yields a substantial performance improvement in comparison to simple or non-replicated approaches. As such, GlobeDB improves upon other research demonstrating that application-specific replication improves performance as well (e.g., [Gao et al., 2003]). In particular, we argue that there is often no compelling reason to adapt the functional design of an application in order to support replication. By automatically partitioning and replicating an application's associated database, we can achieve the same results without involving the application designer or requiring other human intervention.

The rest of the chapter is organized as follows. Section 4.2 presents several design issues involved in building the system and motivates our design choices. Section 4.3 presents GlobeDB's architecture and Section 4.4 describes the design of the data driver, the central component of GlobeDB. Section 4.5 describes GlobeDB's replication and clustering algorithms. Section 4.6 presents an overview of our prototype implementation and its internal performance. Section 4.7 presents the relative performance of GlobeDB and different edge service architectures for the TPC-W benchmark. Finally, Section 4.8 discusses the related work and Section 4.9 concludes the chapter.

4.2. DESIGN ISSUES

To illustrate the benefits of autonomic replication, consider the scenarios presented in Figure 4.1 that show edge server(s) hosting a Web application. As seen in the figure, there is a fraction of data accessed only by clients of server 1, another fraction by clients of server 2 and the rest are accessed by clients of both servers. Not replicating at all (centralized system) can result in poor client response time. Replicating all data everywhere (full replication) can result in significant update

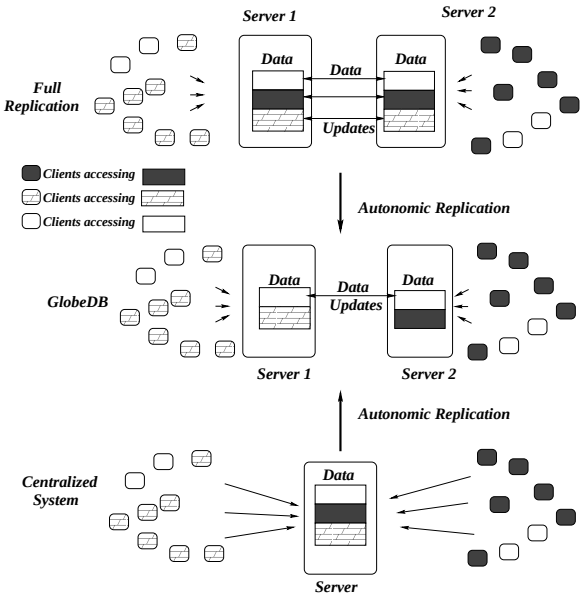


Figure 4.1: Example of benefits of autonomic replication.

traffic between servers for data they barely access. In such scenarios, GlobeDB can be very useful as it places the data in only those servers that access them often. This can result in a significant reduction in update traffic and improved client-perceived response time.

Building a system for autonomic replication of Web application data requires addressing many issues such as identifying the granularity and constituents of the data segments, finding the optimal placements for each data unit, and maintaining consistency of replicated data units. This section discusses these issues in detail.

4.2.1. Application transparency

An important issue is to decide the extent to which an application should be aware of data replication. Replication can yield the best performance if it is completely tuned to the specific application and its access patterns [Gao et al., 2003]. However, this requires significant effort and expertise from an application developer. As a consequence, optimal performance is often not reached in practice. Furthermore, changes in access pattern may warrant changes in in the replication strategies, thereby making the design of an optimal strategy even more complex.

In our system, we chose a *transparent* replication model. The application developer need not worry about replication issues and can just stick to the functional aspects of the application. The system will automatically find a placement and replication strategy, and adapt it to changing access patterns when needed.

4.2.2. Data granularity

The underlying principle behind our system is to place each data unit only where it is accessed. In our previous research on replication for static Web pages, we showed that the optimal replication performance in terms of both client-perceived latency and update bandwidth can be achieved if each Web page is replicated according to its individual access patterns [Pierre et al., 2002]. A naive transposition of this result would lead to replicating each database record individually. However, such fine-grained replication can result in significant overhead as the system must maintain replication information for each record.

In our system, we employ an approach where the data units are initially defined at a very fine grain. Data units having similar access patterns are then automatically clustered by the system. The system subsequently handles replication at the cluster level, thereby making the problem of managing a cluster feasible without losing the benefits of partial replication. However, a caveat of this approach is that if the access patterns change, then the system must perform re-clustering to sustain good performance. More information on the clustering algorithm used in our system is presented in Section 4.5.

4.2.3. Consistency

One important issue in any replicated system is consistency. Consistency management has two main aspects: update propagation and concurrency control. In update propagation, the issue is to decide which strategy must be used to propagate updates to replicas. GlobeDB uses a *push* strategy where all updates to a data unit are pushed to the replicas immediately. Pushing data updates immediately ensures that replicas are kept consistent and that the servers hosting replicas can serve read requests immediately.

Yet, propagating updates is not sufficient to maintain data consistency. The system must also handle concurrent updates to a data unit emerging from multiple servers. Traditional non-replicated DBMSs perform concurrency control using locks or multiversion models [Gray and Reuter, 1993]. For this, a database requires an explicit definition of *transaction*, which contains a sequence of read/write operations to a database. For example, PostgreSQL uses a variant of multiversion model called snapshot isolation to handle concurrent transactions.[1] When querying a database each transaction sees a snapshot of consistent data (a database version), regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data produced by concurrently running update transactions.

[1] http://www.postgresql.org/

However, concurrency control at the database level can serialize updates only to a single database server and does not handle concurrent updates to replicated data units at multiple edge servers. Traditional solutions such as two-phase commit are rather expensive as they require global locking of all databases, thereby reducing the performance gains of replication. To handle this scenario, the system must serialize concurrent updates from multiple locations to a single location. GlobeDB does not guarantee full transaction semantics for the entire database, but enforces consistency for each data unit individually. In other words, it imposes that each transaction be composed of a single query/operation which at most modifies a single data unit. Note that the granularity of updates can be extended to a data cluster level as all data items within a single cluster are always replicated together. The concurrency control adopted in our system is explained in Section 4.3.

4.2.4. Data placement

Automatic data placement requires the system to find a set of edge servers to host the replicas of a data cluster according to certain performance criteria. One can measure the system performance by metrics such as average read latency, average write latency, amount of update traffic, etc. But a naive approach based on optimizing the system performance for one of these metrics *alone* can easily result in degrading the performance according to other metrics. For example, a system can be optimized for minimizing read latency by replicating the data to all replica servers. However, this can lead to huge update traffic if the number of updates is high.

In general, there is a clear tradeoff between the performance gain due to replication and the performance loss due to consistency enforcement. However, there is no universal definition of "best" tradeoff. In fact, each system administrator should specify a particular tradeoff based on the system needs. For example, the administrator of a CDN with (theoretically) unlimited bandwidth may choose to optimize on client response time alone. The same administrator, when facing a bottleneck at the central server, may prefer to minimize update traffic.

In our system, the system administrator specifies relative performance tradeoffs as the weights of a *cost function*. This function aggregates multiple performance metrics into a single abstract metric. Optimizing the cost function is equivalent to optimizing the global system performance. This function therefore acts as a measure of the desired system performance and aids the system in making its placement decisions. The cost function and placement algorithms are presented in Section 4.5.

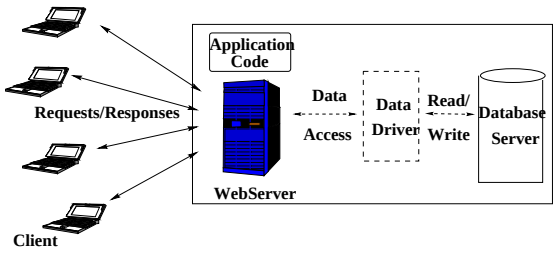


Figure 4.2: Application Model

4.3. SYSTEM ARCHITECTURE

4.3.1. Application model

The application model of our system is shown in Figure 4.2. An application is made of code and data. The code is written using standard dynamic Web page technology such as PHP and is hosted by the Web server (or the Web application server). It is executed each time the Web server receives an HTTP request from its clients, and issues read/write accesses to the relevant data in the database to generate a response.

Access to the data is done through a data driver which acts as the interface between code and data.[2] The data driver preserves distribution transparency of the data and has the same PHP interface as regular PHP drivers. It is responsible for finding the data required by the code, either locally or from a remote server, and for maintaining data consistency.

We assume that the database is split into n data units, D_1, D_2,$\cdots$, D_n, where a data unit is the smallest granule of replication. Each unit is assumed to have a unique identifier, which is used by the data driver to track it. An example of a data unit is a database record identified by its primary key.

GlobeDB enforces consistency among replicated data units using a simple master-slave protocol: each data cluster has one master server responsible for serializing concurrent updates emerging from different replicas. GlobeDB assumes that each database transaction is composed of a single query which modifies at most a single data unit. When a server receives an update request, it forwards the request to the master of the cluster, which processes the update request and propagates the result to the replicas. Note that this model is sometimes called eventual consistency. If stronger consistency is needed, then models such as session guarantees can be envisaged [Terry et al., 1994].

[2]This driver is different from conventional PHP drivers as it not just an interface driver but also responsible for functional aspects such as location of replicated data.

4.3.2. System architecture

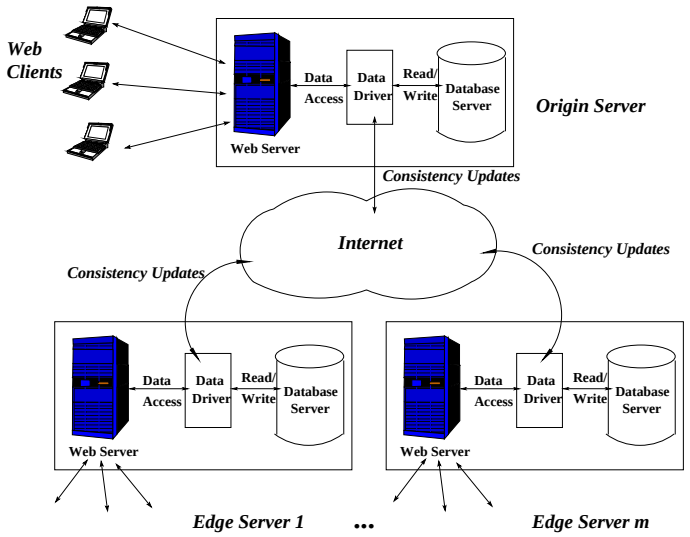


Figure 4.3: System Architecture - Edge servers serving clients close to them and interactions among edge servers goes through Wide-area network.

The architecture of GlobeDB is presented in Figure 4.3. An application is hosted by *m* edge servers spread across the Internet. Communication between edge servers is through wide-area networks incurring wide-area latency. Each client is assumed to be redirected to its closest edge server using enhanced DNS-based redirection [Dilley et al., 2002; Fei et al., 1998]. Furthermore, for each session, a client is assumed to be served by only one Web server.

When a client issues an HTTP request to a Web server, the request is forwarded to the application code (e.g., PHP) residing in the server. The application code usually issues a number of read/write accesses to its data through the data driver. The application data are partially replicated, so the local database hosts only a subset of all data clusters. The data driver is responsible for finding the relevant data either locally or from a remote edge server if the data are not present locally. Additionally, when handling write data accesses, the driver is also responsible for ensuring consistency with other replicas.

To perform autonomic replication, the system must decide on the placement of replicas for each data cluster and choose its master according to its access pattern. To this end, each application is assigned one *origin server*, which is responsible for making all application-wide decisions such as clustering data units and placing clusters on edge servers. The origin server performs replica placement periodi-

cally to handle changes in data access patterns and the creation of new data units. For reasons explained below, the origin server also has a full database replica.

Consistency is enforced using a simple master-slave protocol: each data cluster has one master server responsible for serializing concurrent updates emerging from different replicas. When a server holding a replica of a data cluster receives a read request, it is answered locally. When it receives an update request, it is forwarded to the master of the data cluster. If the server does not have a replica, then requests are forwarded to the origin server. More information on locating data units is presented in Section 4.4.

GlobeDB does not support the notion of transaction boundaries (where multiple read and write queries are grouped together as a single execution unit) as usually defined in the traditional database. GlobeDB handles each read and write query individually and guarantees eventual consistency. Note that eventual consistency does not guarantee that all replicas are identical at all times. The master of a cluster delivers updates to all replicas in the same order without any global locking. This can lead to transient situations where the latest updates have been applied only to a fraction of the replicas. We assume that this can be tolerated by the application as each client session is handled by only one edge server in the network. Note that even existing solutions over commercial database systems such as DBCache [Bornhövd et al., 2004] and MTCache [Larson et al., 2004] face similar issues and guarantee the same level of consistency as GlobeDB.

4.4. DATA DRIVER

The data driver is the central component of GlobeDB. It is in charge of locating the data units required by the application code and maintaining consistency of the replicated data.

4.4.1. Types of queries

The primary functionality of the data driver is to locate the data units required by the application code. Data access queries can be classified into write and read queries based on whether or not an update to the underlying database takes place. Another way of classifying queries is based on the number of rows matched by the query selection criterion. We refer to the queries based on primary keys of a table or that result in an exact match for only one record as *simple queries*. An example of a simple query is "Find the customer record whose userid is 'xyz'." Queries based on secondary keys and queries spanning multiple tables are referred to as *complex queries*. An example of a complex query is "Find all customer records whose location is 'Amsterdam'."

As noted earlier, we assume that each data unit has a unique identifier. For fine-grained data units, such as database records, we use the primary key as the record's unique identifier. This allows the data driver to map simple queries to required data units, which makes locating data units relatively straightforward.

For answering a complex query, the driver cannot restrict its search to its local database but needs to inspect the entire database. Such a query can be answered by forwarding it to a subset of servers that jointly have the complete database table. For the sake of simplicity, we stipulate that the origin server has a replica of all data units. For this reason all complex queries are forwarded to the origin server which will incur a wide-area latency.

At the outset, being able to answer only simple queries locally might appear very limiting. However, typical Web applications issue a majority of simple queries. To get an idea of the percentage of simple queries used on real e-commerce applications, we examined the TPC-W benchmark which models a digital bookstore [Menasce, 2002]. We collected the SQL traces generated by execution of the benchmark's ordering mix workload.[3] We analyzed the accesses to the book tables (which contains the records pertaining information about individual books) and found that more than 80% of the accesses to the table consist of simple queries or can be easily re-written as simple queries. Similarly, about 80% of accesses to the customer tables use simple queries. Similar figures are seen for other workload mixes of TPC-W. In TPC-W, updates to a database are always made using simple query.

4.4.2. Locating data units

The data driver of each edge server maintains three tables. The *cluster-membership* table stores the identifiers of data units contained in each cluster. The *cluster-property* table contains the following information for each data cluster: the origin server, the master replica, and the list of servers that host a copy of this cluster. These two tables are fully replicated at all edge servers. Each driver also maintains an *access table* to keep track of the number of read and write accesses to each cluster.

To answer simple queries, the driver locates a data unit by identifying the cluster to which the data unit belongs, using the *cluster-membership* table. Once the appropriate cluster is identified, the driver uses the *cluster-property* table to find details about the location of the cluster and its master. Upon each read or write access, the driver updates the access table accordingly. The data driver forwards all complex queries to the origin server.

A naive design of the *cluster-membership* table can be a scalability bottleneck.

[3] http://www.ece.wisc.edu/~pharm/tpcw.shtml

```
if complex query then
    Execute query at the origin server and return result;
else
    if read then
        Execute query locally;
        if execution returns result then
            return result;
        else
            execute on origin server and return result;
        end
    else
        Get cluster id of data unit from (local or origin server);
        Find master for cluster from cluster-property table;
        Execute query on master server and return result;
    end
end
```

Algorithm 1: Pseudocode used by data driver for executing queries

In our initial implementation, we used bit arrays for numerical primary key IDs and bloom filters for non-numerical IDs [Bloom, 1970]. A typical bit-array based cluster membership table will have a size of only 125 Kbits for each cluster to represent a database with a million data units. Such a small size allows the table to reside in main memory thereby resulting in faster access. However, these filters have several disadvantages. First, storing non-numerical IDs using Bloom filters can result in potential inaccuracies that result in redundant network traffic. Second, the system needs to allocate enough memory for filters to accommodate creation of new data units in the future, which poses a scalability problem.

To overcome these shortcomings, in our current implementation the *cluster-membership* table is stored along with the respective database records. In this implementation, each database record has an extra attribute that indicates which cluster the record belongs to. The pseudocode for executing queries by the data driver is shown in Algorithm 1. As seen in the figure, simple queries with read accesses are always first executed locally. If a record is returned, the result is returned immediately.[4] Otherwise, the query is forwarded to the origin server (which contains a full copy of the database). For update queries, the driver first runs a query to find the cluster ID. If the data record is replicated locally, its cluster ID will be returned and the information regarding who its master is can be obtained from the *cluster-property* table. Then, the update query is forwarded to the master. In case of update queries to a data unit not present locally, the query to find cluster information for the data unit will return no result and this query is

[4]This is also done for exact match queries (queries that match a single data unit) based on non-primary queries.

then run on the origin server. Subsequently, the update query will be sent to the appropriate master server. The *cluster-property* table is simply implemented as a file. This table is created and updated (whenever necessary) by the origin server and is replicated to the edge servers.

4.5. REPLICATION ALGORITHMS

Replicating an application requires that we replicate its code and data. For the sake of simplicity, in this chapter we assume that the code is fully replicated at all replica servers. In this section, we present the algorithms we use for clustering data units, placing the data cluster, and selecting their master. For the placement of data, we use a *cost function* that allows the system administrator to tell GlobeDB his/her idea of optimal performance. As we explain in detail in this section, GlobeDB uses this function to assess the goodness of its placement decisions.

4.5.1. Clustering

As we mentioned earlier, in GlobeDB, data units with similar access patterns are clustered together. A similar problem was addressed in [Chen et al., 2002b] in the context of clustering static Web pages to reduce the overhead in handling replicas for each Web page. The authors propose several spatial clustering algorithms to group pages into clusters and incremental clustering algorithms to handle the creation of new pages. They show that these clustering algorithms perform well for real-world Web traces. We use similar algorithms for clustering data units.

The origin server is responsible for clustering the data units during the initial stages of system. The origin server collects access patterns of data units from all edge servers into its access table. Each data unit D_i's access pattern is modelled as a $2*m$-dimensional vector, $A_i =< r_{i,1},r_{i,2},\cdots,r_{i,m},w_{i,1},\cdots,w_{i,m} >$, where $r_{i,j}$ and $w_{i,j}$ are respectively the number of read and write accesses made by the edge server R_j to the data unit D_i. The origin server runs a spatial clustering algorithm that uses correlation-based similarity on the access vectors of the data units. As a result, two data units D_i and D_j are grouped into the same cluster if and only if A_i and A_j are similar. In correlation-based similarity, the similarity between two data units D_i and D_j is given by

$$Sim(i,j) = \frac{\sum_{k=1}^{2*m}(a_{i,k}-\bar{a_i})(a_{j,k}-\bar{a_j})}{\sqrt{\sum_{k=1}^{2*m}(a_{i,k}-\bar{a_i})^2 \sum_{k=1}^{2*m}(a_{j,k}-\bar{a_j})^2}}$$

where $< a_{i,1},a_{i,2},\cdots,a_{i,m} > = < r_{i,1},r_{i,2},\cdots,r_{i,m} >$ and $< a_{i,m+1},a_{i,m+2},\cdots,a_{i,2*m} >$

$= < w_{i,1}, \cdots, w_{i,m} >$. Data units D_i and D_j are clustered together if $Sim(i, j) \geq x$, for some threshold value x, where $0 \leq x \leq 1$.

The origin server iterates through data units that are yet to be clustered. If a data unit D_i is sufficiently close to the access vector of an existing cluster, it is merged into it. Otherwise, a new cluster with D_i as the only member is created. Once the data clusters are built, the origin server creates the appropriate cluster-membership table for each cluster. Obviously, the value of x has an impact on the effectiveness of the replication strategy. In all experiments from Section 4.6 we fix the threshold x to 0.95. We will study the process of determining the optimal threshold value in the near future.

Data clustering works well if data units once clustered do not change their access pattern radically. However, if they do, then the clusters must be re-evaluated. The process of re-clustering requires mechanisms for identifying stale data units within a cluster and then re-clustering them. We do not study the issue of (re-)clustering in detail in this thesis. Clustering is orthogonal to the replication strategy and it mainly involves determining when to re-cluster and how to re-cluster efficiently. Also note that re-clustering can be done by progressively invalidating and validating copies at the different edge servers as it is done for data caches. For an in-depth study of different clustering algorithms that can be applied for GlobeDB, we refer interested readers to [Yang, 2005].

4.5.2. Selecting a replication strategy

The origin server must periodically select the best replication strategy for each cluster. A replication strategy involves three aspects: *replica placement*, *consistency mechanism*, and, in our case, *master selection*. As we use push strategy as our consistency mechanism, the selection of a replication strategy for a cluster boils down to deciding about replica placement and selecting the master.

As noted earlier, to select the best replication strategy, the system administrator must specify what "best" actually means. In GlobeDB, we represent overall system performance into a single abstract figure using a *cost function*. A cost function aggregates several evaluation metrics into a single figure. By definition, the best configuration is the one with the least cost. An example of a cost function which measures the performance of a replication strategy s during a time period t is:

$$cost(s,t) = \alpha * r(s,t) + \beta * w(s,t) + \gamma * b(s,t)$$

where r is the average read latency, w is the average write latency, and b is the amount of bandwidth used for consistency enforcement.

The values α, β and γ are weights associated to metrics r, w, and b respectively. These weights must be set by the system administrator based on system constraints and application requirements. A larger weight implies that its associ-

ated metric has more influence in selecting the "best" strategy. For example, if the administrator wants to optimize on client performance and is not concerned with the bandwidth consumption, then weights α and β can be increased. Finding the "best" system configuration now boils down to evaluating the value of the cost function for every candidate strategy and selecting the configuration with the least cost.

In GlobeDB we use the cost function as a *decision making tool* to decide on the best server placements and master server for each cluster. Ideally, the system should periodically evaluate all possible combinations of replica placement and master server configurations for each cluster with the cluster's access pattern for the past access period. The configuration with the least cost should be selected as the best strategy for the near future. This relies on the assumption that the past access patterns are a good indicator for the near future. This assumption has been shown to be true for static Web pages and we expect the dynamic content will exhibit similar behavior [Pierre et al., 2002].

Ideally, the system should treat the master selection and replica placement as a single problem and select the combination of master-slave and replica placement configuration that yields the minimum cost. However, such a solution would require an exhaustive evaluation of $2^m * m$ configurations for each data cluster, if m is the number of replica servers. This makes this solution computationally infeasible. In GlobeDB, we use heuristics to perform replica placement and master selection (discussed in the next subsections). We propose a number of possible heuristics for placement and a method for optimal selection of master server. This reduces the problem of choosing a replication strategy to evaluating which combination of master server and placement heuristics performs the best in any given situation. After selecting the best replica placement and master for each data cluster, the origin server builds the cluster-property table and installs it in all edge servers.

4.5.3. Replica placement heuristics

Proper placement of data clusters is important to obtain good client latencies and reduced updated traffic. We define a family of placement heuristics *Py* where an edge server hosts a replica of a data cluster if its server generates at least *y*% of data access requests.

Obviously, the value of *y* affects the performance of the system. A high value of *y* will lead to creating no replica at all besides the origin server. On the other hand, a low value of *y* may lead to a fully replicated configuration.

Expecting the system administrator to determine the appropriate value for *y* is not reasonable, as the number of parameters that affect system performance is high. Instead, in GlobeDB, administrators are just expected to define their

preferred performance tradeoffs by choosing the weight parameters of the cost function. The origin server will then evaluate the cost value for placement configurations obtained for different values of y (where y=5,10,15,20,25), and select the one that yields the least cost as the best placement configuration. Note that these heuristics are designed primarily to reduce the network latency to fetch a data unit and assumes that the edge servers are well provisioned (using a cluster of physical servers) to handle their query load.

4.5.4. Master selection

Master selection is essential to optimize the write latency and the amount of bandwidth utilized to maintain consistency among replicas. For example, if there is only one server that updates a data cluster, then that server should be selected as the cluster's master. This will result in low write latency and less update traffic as all updates to a cluster are sent to its master and then propagated to the replicas.

We use a method for optimal selection of a master server that results in the least average write-latency. Let $w_{i,j}$ be the number of write access requests received by edge server R_j for cluster i and l_{jk} be the latency between edge server j and k (we assume that latency measurements between servers are symmetric, i.e., l_{kj}=l_{jk}). The average write latency for data cluster i whose master is k is given by: $wl_{ki} = (\sum_{j=1}^{m} w_{i,j} * l_{jk})/(\sum_{j=1}^{m} w_{i,j})$. The origin server selects the server with lowest average write latency as the master for a data cluster .

4.6. IMPLEMENTATION AND ITS PERFORMANCE

In this section, we discuss the salient components of our prototype system and also present performance measurements on the overhead that the data driver introduces to a single edge server.

4.6.1. Implementation overview

The salient components of our prototype system are shown in Figure 4.4. We implemented our data driver by modifying the existing Apache PHP driver for PostgreSQL, by adding new query interfaces to the existing PHP driver. This driver can be added as a module to the Apache Web server.

Each edge server runs a *logger service*, which is responsible for collecting information regarding which cluster was accessed by the Web clients. The logger is implemented as a stand-alone multi-threaded server that collects information from the driver and periodically updates the access database.

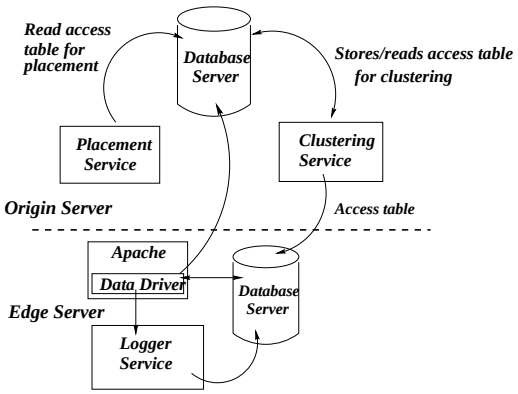


Figure 4.4: Salient components of the system

As seen in the figure, the origin server runs two special services, *clustering* and *replica placement* services. The clustering service performs the clustering of data units during the initial stages of system deployment. It periodically collects access patterns from the edge servers and stores it in its database. Subsequently, it performs clustering with the algorithm described in Section 4.5. The origin server acts only a back-end database so it does not need to have a Web server or logger service.

The *replica placement* service in the origin server finds the "best" locations for hosting a replica and master for each cluster. It does so by evaluating the cost obtained by different replica placement strategies and master selection for the cluster's access pattern in the previous period. Note that once data units are clustered, the logger service in the edge servers starts collecting access patterns at the cluster level. This information is then used by the replica placement service to place the replicas. Upon deciding which servers would host which clusters, the placement service builds the cluster-property table for each cluster and copies it to all edge servers.

To perform periodic adaptation, the replica placement service is invoked periodically. We note that the period of this adaptation must be set to a reasonable value to ensure good performance and stability. We intend to study the need and support for continuous adaptation and effective mechanisms for them in the future.

The clustering service is also run periodically. The bulk of its work is done during the initial stages when it has to cluster large numbers of data units. Later, during every period, it performs incremental clustering of newly created data units. The current prototype does not perform any kind of re-clustering.

4.6.2. Measuring the overhead of the data driver

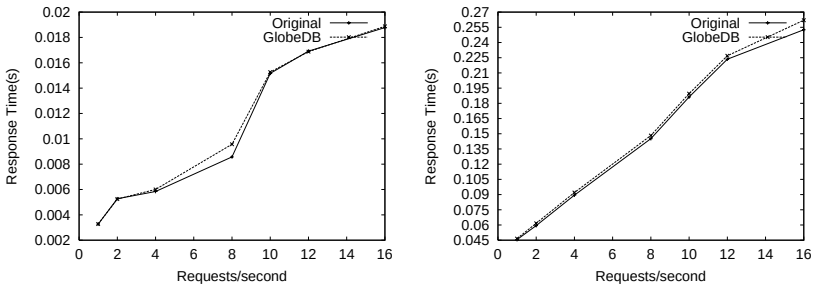
The data driver receives SQL queries issued by the application code and is responsible for locating the relevant data unit from the local server or from a remote server. It does so by checking appropriate cluster-membership tables for each request. Performance gains in terms of latency or update traffic occur when the clusters accessed by a given edge server can be found locally. However, it is important to ensure that the gain obtained by replication is not annulled by the overhead due to checking the cluster-membership table for each data access. To analyze this, we study the response time of our driver when executing a query on a local replicated data in comparison to the response time of the original PHP driver.

In our experimental setup, we ran the Apache Web server on a P*III*-900MHz Linux machine with 1 GB main memory. The PostgreSQL database also runs on an identical machine. We created a book table with fields such as book id, book name, author id, stock quantity and five other integer fields. The database was populated with $100,000$ random records.

We measured the execution latencies of read and write queries using the original PHP driver and the GlobeDB PHP driver for different throughput values. In both cases, the requested data is available locally; the only difference is that the GlobeDB driver needs to check its cluster membership and cluster-property tables before each data access. Read queries read a random database record using a simple query. Write queries increment the stock field of a randomly chosen book record. To make sure that each access is local, the server is assumed to be the master for all clusters. We computed the execution latency as the time taken by the server to generate the response for a request. We do not include the network latency between the client and server as the objective of this experiment is only to measure the overhead of our driver in processing the request.

The results of this experiment are given in Figure 4.5. Even for high throughputs, the overhead introduced by our implementation is between 0 and 5 milliseconds for read accesses, and at most 10 milliseconds for write accesses. This is less than 4% of the database access latencies incurred by the original driver.

We conclude that the overhead introduced by our driver is very low and, as we shall see in the next section, negligible compared to the wide-area network latency incurred by traditional non-replicated systems. Clearly, this experiment only shows that the overhead introduced by GlobeDB's driver is low and does not give any insights on the performance of GlobeDB's replication mechanisms. This is the focus of the experiments presented in the subsequent section.



(a) Read Accesses **(b)** Write Accesses

Figure 4.5: A comparative study of GlobeDB driver implementation with original PHP driver for reading and updating local data units

4.7. PERFORMANCE EVALUATION: TPC-W BOOKSTORE

The experiments presented in the previous section show that the overhead introduced by GlobeDB's driver in a single edge server is low. However, it does not offer any insight into the performance gains of GlobeDB. In this section, we study the performance gain that could be obtained using GlobeDB while hosting an e-commerce application. We chose the TPC-W benchmark and evaluated the performance of GlobeDB in comparison to other existing systems for different throughput values over an emulated wide-area network. As the experiment results presented in this section will show, GlobeDB can reduce the client access latencies for typical e-commerce applications with large mixture of read and write operations without requiring manual configurations or performance optimizations.

4.7.1. Experiment setup

We deployed our prototype across 3 identical edge servers with Pentium III 900 Mhz CPU, 1-GB of memory and 120GB IDE hard disks. The database servers for these edge servers were run on separate machines with the same configuration. Each edge server uses Apache 2.0.49 Web servers with PHP 4.3.6. We use PostgreSQL 7.3.4 as our database servers. The origin server uses an identical configuration as the edge servers except that it acts just as a backend database and does not run a Web server. We emulated a wide-area network (WAN) among servers by directing all the traffic to an intermediate router which uses the NIST Net network emulator.[5] This router delays packets sent between the different servers to simulate a realistic wide-area network. In the remaining discussion, we refer to links

[5] http://snad.ncsl.nist.gov/itg/nistnet/

via NISTNet with a bandwidth of 10*Mbps* and a latency of 100ms as WAN links and 100*Mbps* as LAN links. We use two client machines to generate requests addressed to the three edge servers. A similar setup to emulate a WAN was used in [Gao et al., 2003].

We deployed the TPC-W benchmark in the edge servers. TPC-W models an on-line bookstore and defines workloads that exercise different parts of the system such as the Web server, database server etc. The benchmark defines activities such as multiple on-line browsing sessions using Remote Browser Emulators (RBEs), dynamic page-generation from a database, contention of database accesses and updates. The benchmark defines three different workload scenarios: browsing, shopping and ordering. The browsing scenario has mostly browsing related interactions (95%). The shopping scenario consists of 80% browsing related interactions and 20% shopping interactions. The ordering scenario contains an equal mixture of shopping and browsing interactions. In our experiments, we evaluate GlobeDB for the ordering scenario, as it generates the highest number of updates. The performance metrics of the TPC-W benchmark are WIPS (Web Interactions Per Second), which denotes the throughput one could get out of a system and WIRT, which denotes the average end-to-end response time that would be experienced by a client.

The benchmark uses the following database tables: (i) *tpcw_item* stores details about books and their stocks; (ii) *tpcw_customer* stores information about the customers; (iii) *tpcw_author* stores the author-related information; (iv) *tpcw_order* and *tpcw_orderline* store order-related information. In our experiment, we study the effects of replication only for the *tpcw_customer* and the *tpcw_item* tables as these are the only tables that receive updates and read accesses simultaneously. The *tpcw_author* table is replicated everywhere as it is never updated by Web clients.

In our experiments, we want to compare the performance of GlobeDB with traditional centralized and fully replicated scenarios. In principle, a fully replicated system should replicate all tables at all edge servers. However, each record of ordering-related tables is mostly accessed by only a single customer. In addition, the entire order database is used to generate the "best sellers" page. In such a scenario, full replication of ordering-related tables would be an overkill as it would result in too much update traffic among edge servers. Any reasonable database administrator would therefore store ordering-related database records only in servers that created them and maintain a copy at the origin server (which would be responsible for generating responses to "best sellers" queries). So, for a fair comparison with GlobeDB, we implemented this optimization manually for the fully replicated system.

We use the open source PHP implementation of TPC-W benchmark.[6] We disabled the image downloading actions of RBEs as we want to evaluate only the response time of dynamic page generation of the edge server. To account for the geographical diversity of the customers, we defined three groups of clients which respectively issue their requests to a different edge server. We believe this is a realistic scenario as customers typically do not move often and are usually redirected to the server closest to them.

The *tpcw_item* table stores fields such as its unique integer identifier, author identifier, item name, price and stock. In addition to these fields, it also stores five integer fields that have identifiers of books that are closely related to it and have a similar customer base. These fields are read by the application code to generate promotional advertisements when a client reads about a particular book. For example, related fields of a Harry Potter book may contain identifiers of the other Harry Potter books. In our experiment, we filled these related identifiers as follows: the item records are classified into three groups and related entries of each item is assigned to an item in the same group. In this way, the related field truly reflects books with similar customer base. Furthermore, TPC-W stipulates that each Web session should start with a book for promotion. In our experiments, clients of edge server *i* receive the starting random book *id* only from item group *i*. At the outset, this might look as if each client group request books from only one item group. However, this is not the case as the RBE clients select the books to view also from other interactions, such as best-sellers and search result interactions, which have books spanning across multiple item groups.

Even though the assumptions we make about the clients access patterns are realistic, they do not capture all kinds of client access patterns. In one of our earlier studies, we simulated our proposed replication techniques for different kinds of access patterns from uniform popularity (all clients are interested in all data) to a very skewed popularity (only a small set of clients are interested in a particular piece of data) using statistical distributions [Sivasubramanian et al., 2004a]. We found that our techniques perform well in all cases compared to traditional fully replicated and centralized systems. Hence, in this experiment we restrict our evaluations to only the workload described in this setup and study the relative performance of different system implementations.

In our experiments, for each run we ran the benchmark for 8 hours. After this, the origin server collected the access patterns from the edge servers and performed clustering and replication. Analysis of the *tpcw_customer* table resulted in three clusters. Each of these clusters were mostly accessed by only one edge server (different one in each case). Analysis of *tpcw_item* accesses led to four clusters. Three out of the four clusters are characterized by accesses predominantly from

[6] http://pgfoundry.org/projects/tpc-w-php/

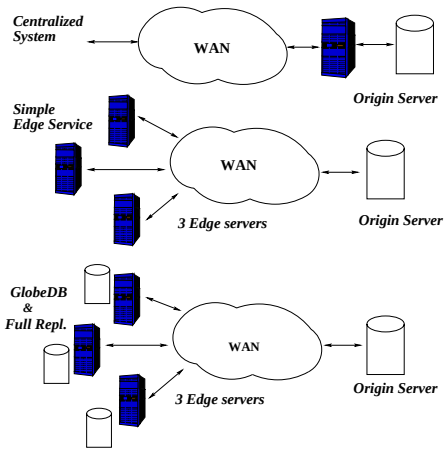


Figure 4.6: Architectures of different systems evaluated

only one edge server (a different one in each case). However, the fourth cluster represents data units that are accessed by clients of all the three edge servers.

4.7.2. Experiment results

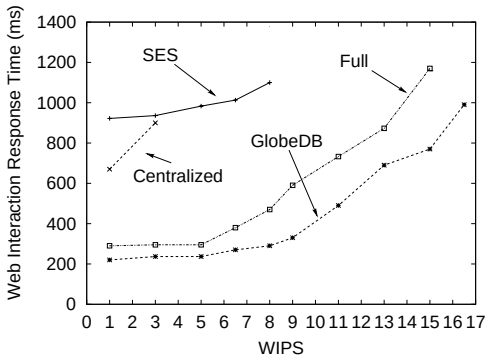


Figure 4.7: Performance of different system architectures running TPC-W benchmark

We evaluated the performance of four different systems: (i) a traditional centralized architecture where the code and the database is run in a server across a WAN (Centralized), (ii) a simple edge server architecture where three edge servers run a Web server and the database is run in a server across the WAN (SES), (iii) our proposed system with 3 edge servers (GlobeDB) and (iv) a full replication

system (Full) which is similar to the GlobeDB setup - the only difference being that the *tpcw_item* and *tpcw_customer* tables are fully replicated at all edge server unlike GlobeDB.

As we noted earlier, replication decisions are made through evaluation of the cost function and its weights α, β and γ as described in Section 4.5. In our experiments, we assumed the system administrator wants to optimize the system for improved response time and assigned higher weights to α and β compared to γ. We set $\alpha=2/r_{max}$; $\beta=2/w_{max}$; and $\gamma=1/b_{max}$, where r_{max}, w_{max} and b_{max} are maximum values of average read latency, write latency, and number of consistency updates, respectively. These values effectively tell the system to consider client read and write latency to be twice as important in comparison to update bandwidth (on a normalized scale). These weights result in the following placement configuration: For the three clusters of the customer table, three different placement configurations are obtained where a different edge server is the master while hosting a replica of the cluster. Similar placement configurations are obtained for three out of the four clusters of the book table. The fourth cluster (which contains data units that were accessed from all edge servers) was automatically placed at all the three edge servers as it represents book records popular among all clients.

In our experiments, we study the WIRT for different WIPS until the database server cannot handle more connections. The results of our experiments are given in Figure 4.7. Even for low throughputs GlobeDB performs better than the traditional Centralized and SES architectures and reduces response time by a factor 4. GlobeDB performs better than SES and a centralized system as it is able to save on wide-area network latency for each data access because it is able to find many data records locally. Moreover, this shows that replicating application code is not sufficient to obtain good performance.

The difference in WIRT between the GlobeDB and Full setup varies from 100 to 400ms. This is because the GlobeDB system is capable of performing local updates (the server that writes most to a database cluster is elected as its master) but the Full setup forwards all updates to the origin server. These updates constitute 30% of the workload. On the other hand, the Full setup gains in the fact it can handle some complex queries such as search result interactions locally, while GlobeDB forwards it to the origin server.

It is obvious that the fully replicated system produces more update traffic than GlobeDB as it propagates each update to all edge servers. In this experiment, we found that GlobeDB reduces the update traffic by a factor 6 compared to Full replication. This is because GlobeDB prevents unnecessary update traffic by placing data only where they are accessed. Reducing the update traffic potentially leads to significant cost reduction as CDNs are usually charged by the data centers for the amount of bandwidth consumed.

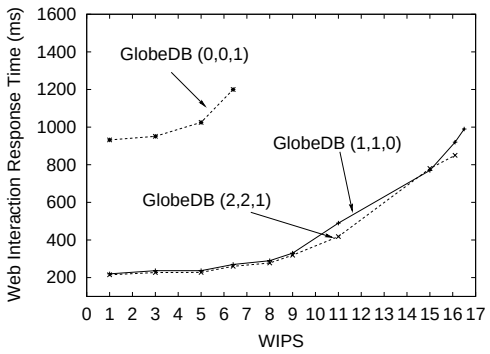


Figure 4.8: Relative Performance of Autonomic System architectures

The centralized and SES systems pay the penalty for making connections over a WAN link. Note that between these systems the Centralized setup yields lower response time as each client request travels only once over the WAN link. In the SES setup, each Web request triggers multiple data accesses that need to travel across WAN links thereby suffering huge delays. However, among these traditional systems, the SES architecture yields better throughput as it uses more hardware resources, i.e., the Web server and the database system are running in separate machines. SES yields a throughput of 7.9 req/s, while the centralized architecture yields much less (1.9 req/s) as it runs both Web server and database on the same machine.

Replication also affects throughput. GlobeDB attains a throughput of 16.9 req/s and is 2 WIPS better than the Full setup and 8 WIPS better than SES. It performs better than SES because the data handling workload is shared among many database servers. While the Full setup has the same number of database servers as GlobeDB, the latter sustains higher throughput as a server does not receive updates to data that it accesses rarely. This reduces the workload of individual database servers in addition to reducing overall consumed bandwidth.

In conclusion, the results of the experiments show that the GlobeDB's automatic data placement can reduce client access latencies for typical e-commerce applications with a large mixture of read and write operations. With these experiments we have shown that automatic placement of application data is possible with very little administration. We believe this is a promising start for realizing a truly scalable self-managing platform for large-scale Web applications.

4.7.3. Effect of the cost function

In our earlier experiments, we showed that autonomic placement of data can yield better performance than traditional strategies. As we noted in Section 4.5, the underlying decision making tool used for autonomic placement is the *cost function* and its weights α, β, and γ.

The objective of the results presented in this section is not to show what are the right weights for a system. Rather, this experiment is just a guide to show the utility of our cost function and the simplicity with which it can attain different objectives (latency optimization, bandwidth optimization or a mixture of it).

We evaluated the relative performance of autonomic systems that uses three different weight parameters, in effect three different criteria for placement. The three systems evaluated are: (i) GlobeDB(0, 0, 1): a system with weights (α, β, γ) =(0,0,1), which implies the system wants to preserve only the bandwidth and does not care about latency, (ii) GlobeDB (1, 1, 0): a system whose weights are set such that the system cares only about the client latency and does not have any constraints on the amount of update bandwidth. Effectively, this situation leads to creating more replicas. (iii) GlobeDB(2, 2, 1): a system that prefers to optimize latency twice as much as the update bandwidth.

As can be seen, these systems are designed for different conditions. For example, GlobeDB(0, 0, 1) is useful for a system that cannot afford to pay for the wide-area bandwidth consumed in its remote edge servers and GlobeDB(1, 1, 0) is useful for a CDN that values its client quality of service (QoS) more than its maintenance costs consumed for maintaining consistency among replicas.

The goal of this experiment is to analyze the impact of different cost function parameters on the client response time of the TPC-W benchmark and the results are given in Figure 4.8. As seen in the figure, GlobeDB(0, 0, 1) performs the worst in terms of WIRT, as it leads to a placement where the data are not replicated at all. Furthermore, its throughput is saturated because all transactions are sent to a single database server. The other two systems perform equally well and yield good throughput. With respect to update traffic, GlobeDB(2, 2, 1) performs better than GlobeDB(1, 1, 0) and reduces update traffic by a factor of 3.5. Note that, while GlobeDB(1, 1, 0) and a fully replicated system have similar goals, the former yields better WIRT as it is able to perform local updates.

4.8. RELATED WORK

As noted in previous chapters, edge computing systems are not suited for data-intensive Web applications. For such applications, the database often turns out to be the bottleneck. Commercial database caching systems such as DB-

Cache [Bornhövd et al., 2004] and MTCache [Larson et al., 2004] cache the results of selected queries and keep them consistent with the underlying database. Such approaches offer performance gains provided the data accesses contain few unique read and/or write requests. However, the success of these schemes depends on the ability of the database administrator to identify the right set of queries to cache. This requires a careful manual analysis of the data access patterns to be done periodically. Query caching systems such as [Amza et al., 2005], DBProxy [Amiri et al., 2003a],and GlobeCBC perform well only if the query locality is high. As we showed in our evaluations in the previous chapter, applications with poor query locality can benefit from data replication and GlobeDB is suited for these kinds of applications.

In [Gao et al., 2003], the authors propose an application-specific edge service architecture, where the application itself is supposed to take care of its own replication. In such a system, access to the shared data is abstracted by object interfaces. This system aims to achieve scalability by using weaker consistency models tailored to the application. However, this requires the application developer to be aware of an application's consistency and distribution semantics and to handle replication accordingly. This is in conflict with our primary design constraint of keeping the process of application development simple. Moreover, we demonstrated that such an awareness need not be required, as distribution of data can be handled automatically.

Traditional database replication systems such as Postgres-R [Kemme and Alonso, 2000] and MySQL cluster[7] replicate the entire database across servers within a cluster environment. Other database replication middleware systems, such as C-JDBC [Cecchet, 2004], Ganymed [Plattner and Alonso, 2004], [Amza et al., 2003], [Lin et al., 2005] and [Chen et al., 2006], perform similar replication at the middleware level. The focus of these works is to improve the throughput of the underlying backend database within a cluster environment, while the focus of GlobeDB is to improve the client-perceived performance and reducing wide-area update traffic. We note that these works can be combined with GlobeDB to scale the database server at both the edge and the origin.

4.9. CONCLUSION

In this chapter, we presented GlobeDB, a system for hosting Web applications that performs efficient autonomic replication of application data. We presented the design and implementation of our system and its performance. The goal of GlobeDB is to provide data-intensive Web applications the same advan-

[7] http://www.mysql.com/products/database/cluster/

tages CDNs offered to static Web pages: low latency and reduced update traffic. We demonstrated this with experimental evaluations of our prototype implementation running the TPC-W benchmark over an emulated wide-area network. In our evaluations, we found that GlobeDB significantly improves access latencies and reduces update traffic by a factor of 6 compared to a fully replicated system. The major contribution of our work is to show that the process of application development can be largely automated and in such a way that it yields substantial improvement in performance.

GlobeDB faces a throughput bottleneck as all the complex queries and updates need to processed by the origin server. Traditional database replication solutions can help here by distributing the read workload across a cluster of replicas at the origin server. However, traditional database replication algorithms used by most databases still face a throughput bottleneck as they require to apply all update, insertion and deletion (UDI) queries to every database replica. The system throughput is therefore limited to the point where the quantity of UDI queries alone is sufficient to overload one server, regardless of the number of machines employed [Fitzpatrick, 2004]. This observation forms the motivation behind our database replication technique, GlobeTHR, presented in the next chapter.

CHAPTER 5

GlobeTP: Template-Based Database Replication for Scalable Web Applications

5.1. INTRODUCTION

In the previous chapters, we have looked at different database caching and replication systems that are built for scalable hosting of Web applications. However, although these techniques can be very effective depending on the application, their ultimate scalability bottleneck resides in the throughput of the origin database where the authoritative version of the application state is stored. For instance, in GlobeCBC, if the application's query locality is poor then very few database accesses are handled at the edge thereby making origin server the bottleneck. Similarly, in GlobeDB, all complex queries are handled at the origin. So, if a Web application has a large fraction of complex queries, the origin server becomes the bottleneck.

Database replication techniques can of course help here, but the generic replication algorithms used by most databases do not scale linearly as they require to apply all update, deletion and insertion (UDI) queries to every database replica. The origin server's throughput is therefore limited to the point where the quantity of UDI queries alone is sufficient to overload one server, regardless of the number of machines employed [Fitzpatrick, 2004]. The only solutions to this problem are to increase the throughput of each individual server or to use partial replication so that UDI queries can be executed at only a subset of all servers. However, partially replicating a database is in turn difficult because queries can potentially span data items which are stored at different servers. Current partially replicated solutions rely on either active participation of the application programmer (e.g., [Gao et al.,

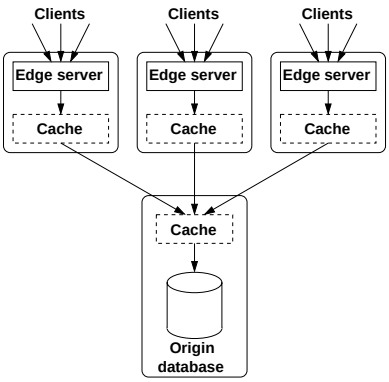


Figure 5.1: Typical Edge-Server Architecture.

2003]) or one special server holding the full database to execute complex queries (e.g., GlobeDB).

In this chapter, we present GlobeTP, a database replication system that exploits the fact that the database queries issued by typical Web applications belong to a relatively small number of query templates. Prior knowledge of these templates allows one to select database table placements such that each query template can be treated locally by at least one server. We demonstrate that careful table placements based on the data span and the relative execution costs of different templates can provide major scalability gains in terms of sustained throughput. We further show that this technique can easily be used in combination with any existing template-based database query caching system, thereby obtaining reduced access latency and yet some more throughput scalability.

The rest of this chapter is organized as follows: Section 5.2 presents the related work. Section 5.3 discusses our system model, and Section 5.4 details our table placement algorithms. Section 5.5 presents performance results and Section 5.6 discusses a number of issues that arise from our approach. Finally, Section 5.7 concludes the chapter.

5.2. BACKGROUND AND RELATED WORK

A typical edge-server architecture followed by many advanced systems to host Web applications is depicted in Figure 5.1. Client requests are issued to edge servers located across the Internet. Each edge server has a full copy of the application code but no database. Database queries are sent out to a local query cache that can answer previously issued queries without incurring wide area latency. Cache misses and UDI queries are issued to the origin server. Queries are then

potentially intercepted by another cache and, in the case of another miss, reach the origin database server to be processed. This architecture has been defined in many versions depending on the specifics of each system.

The first type of edge-server architecture is edge computing infrastructures, where the application code is replicated at all edge servers and no cache is present (see Chapter 2). As noted earlier, the centralization of the database limits the scalability of the approach as the origin database remains the bottleneck.

To remove this bottleneck, various techniques have been proposed to cache the results of database queries at the edge servers (see Chapters 2 and 3). As described in the previous chapters, database caching techniques reduce the database query latency as a number of queries can be answered locally. The total system throughput is also increased because less queries are addressed to the origin server. However, database caching systems have good hit ratio only if the database queries exhibit high temporal locality and contain relatively few updates.

As described in Chapter 2, a common technique used to improve the performance of a database is replication (e.g., Postgres-R [Kemme and Alonso, 2000], C-JDBC [Cecchet, 2004], Ganymed [Plattner and Alonso, 2004], [Amza et al., 2003], [Lin et al., 2005] and [Chen et al., 2006]). Database replication improves throughput as the incoming read query workload is shared among multiple servers. However, if the workload contains a significant fraction of UDI queries, then these systems incurs a limited throughput as all UDI queries must be applied to all replicas. As we show in our experiments, when the load of UDI queries alone is sufficient to overload any one of the servers, then the system cannot improve its throughput any more.

To reduce the overhead of processing UDI workload, some databases such as Oracle use log shipping wherein only the primary executes the UDI query and ships the update logs to the secondary replicas. Usually, the overhead in applying the update logs is cheaper than executing the update query. However, again here primary becomes the throughput and availability bottleneck as it has to handle all the UDI workload.

In [Gao et al., 2003], the authors propose an edge computing infrastructure where the application programmers can choose the data replication and distribution strategies that are best suited for the application. This approach can yield considerable gains in performance and availability, provided that the selected strategies are well suited for the application. However, coming up with the right strategies requires significant insight of the application programmers in domains such as fault-tolerance and weak data consistency. Contrary to this approach, we strive for requiring minimum support from the application programmers, and try to keep replication as transparent to the application as possible. Also the main focus of GlobeTO is performance. We however return briefly to this issue in Section 5.6.2

to show how availability constraints can be taken into account in GlobeTP.

In the previous chapter, we showed how GlobeDB's partial database replication can be used to improve the performance of Web applications. However, GlobeDB's architecture relies on record-level replication granularity. This design choice offers excellent query latency, but does not improve on throughput as a central server must maintain a copy of the full database (and therefore constitutes the throughput bottleneck of the system). Note that replication for throughput and replication for latency do not contradict each other. We show in Section 5.5.5 how GlobeTP can easily be coupled with GlobeCBC so that both throughput and latency can be improved at the same time.

5.3. SYSTEM MODEL

5.3.1. Application model

Web applications are usually implemented by some business logic running in an application server, which is invoked each time an HTTP request is received. This business logic, in turn, may issue any number of SQL queries to the underlying database. Queries can be used to read information from the database, but also to update, delete or insert information. We refer to the latter as UDI queries. We assume that the queries issued by a given Web application can be classified as belonging to a relatively small number of query templates.

The explicit definition of query templates is at the basis of several database caching systems as it allows an easy definition of cache invalidation rules (e.g., GlobeCBC, DBProxy). In contrast, GlobeTP uses the same notion of query templates but in a different manner: it exploits the list of templates to derive table placements that guarantee that at least one server is able to execute each query from the application.

5.3.2. System model

The aim of the work presented in this chapter is to increase the scalability of the origin database depicted in Figure 5.1 in terms of the maximum throughput it can sustain, while maintaining reasonable query execution latency. As shown in Figure 5.2, in our system the origin database is implemented by an array of database servers. We assume that all origin database servers are located within a single data center. The replication granularity in our system is the database table, so each database server hosts a replica of one or more table(s) from the application.

Since not all servers contain all the data, it is necessary to execute each query at a server that has all the necessary data locally. This is ensured by the *query*

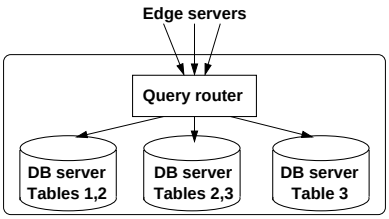


Figure 5.2: Architecture of a partially replicated origin server.

router, which receives all incoming queries and routes each query to a server that contains all the necessary tables to answer the query. To this end, the query router knows the current placement of tables onto databases servers. It is also in charge of maintaining the consistency of the replicated databases. It issues UDI queries to all the servers that hold the tables to be modified; if all queries are successful then the operation is committed, otherwise it is rolled back.

In our implementation, all read and write queries are first received by the query router which in turn executes the query at appropriate replica. Since all UDI queries are queued in a single location at the query router, the router serves a serialization point and maintains consistency across replicas. Note that the current implementation of our system does not support transactions. We believe that this is not a major restriction, as most Web applications do not require transactional database behavior. However, should transactions be required, adding support for them in our system would be relatively straightforward. Since the query router receives all incoming queries before they are executed, it can also act as the transaction monitor and implement any classical protocol such as two-phase commit.

In this chapter, we focus on the structure of a Web application's origin database. Consequently, we make no assumption about the origin of the queries addressed to our system. In the simplest setup, queries can be issued directly by one or more application server(s). However, our system can also be easily coupled with a distributed database query cache such as GlobeCBC. In this case, the same definition of query templates can be used both by the caching system in order to maintain consistency, and the origin database in order to optimize throughput.

5.3.3. Issues

GlobeTP's design is motivated by the observation that the explicit definition of query templates of a Web application allows to select the placement of partially replicated data such that the total system throughput is optimized. We consider that such knowledge allow us to avoid two pitfalls that generic replicated databases necessarily face. First, application-unaware database systems do not know in advance the full set of query templates that will be issued to them. In particular, this

means that it is impossible for them to determine a priori which tables must be kept together, and which ones can be separated. Generic database systems usually address this issue by supporting only full replication, so that the data necessary to answer any query are always available at the same place. However, this has an important impact on the system's throughput. Second, the middleware that determines which replica should treat each read query does not take query characteristics into account. However, the execution times of different queries issued by a given Web application may vary by several orders of magnitude. In such a context, simple round-robin algorithms may not lead to optimal load balancing as shown in [Amza et al., 2003].

To be able to determine the placement of database tables on replica servers that allows to sustain the highest throughput, we must solve three main issues. First, not all possible placements of tables onto servers will allow to find at least one server capable of executing each of the application's query templates. We therefore need to analyze the set of query templates to determine a subset of placements that are functionally correct. Second, we must take the respective query execution times of different templates and their classification as read or UDI queries to determine the best placement in terms of throughput. Besides requiring accurate estimations of query execution times, finding the optimal placement incurs a huge computational complexity, even for relatively small systems. We therefore need a good heuristic. Finally, once the resulting system is instantiated we need to define a load balancing algorithm that allows the query router to distribute read queries efficiently across the servers that can treat them.

5.4. DATA PLACEMENT

The underlying idea behind our approach is to partially replicate the database so that UDI workload can be split across different replicas. This process involves the following three steps: (i) *Cluster Identification:* the process by which we determine the set of database tables that needs to be replicated together, (ii) *Load Analysis:* the process by which we determine the load received by each of the cluster, and (iii) *Cluster Placement:* determining the placement of the identified clusters across the set of database servers so that the load incurred by each of the database replica is minimized.

5.4.1. Cluster identification

Our system relies on placement of individual tables on database servers to minimize the number of servers that must process UDI queries. However, not all placements are functionally correct as all tables accessed by a query template must be

present in the same server for the query to be executed. The goal of cluster identification is to determine sets of tables that must be placed together on at least one server, such that there is at least one server where each query template can be executed.

We must characterize each query template with two attributes: (i) whether it is a read or a UDI query; (ii) the set of tables (also called table cluster) that it accesses. For instance, in the aforementioned query templates, template $QT1$ will be associated to a single-table cluster $\{book\}$, while $QT2$ will be associated to $\{book, author\}$. Clusters can overlap, as a table can belong to multiple clusters.

The problem of finding functionally correct table placements can then be reduced to a cluster placement problem; any table cluster placement will be functionally correct.

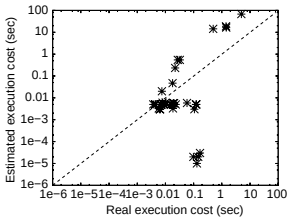
5.4.2. Load analysis

Even though any cluster placement will lead to a functionally correct system, not all placements will lead to the same system throughput. To maximize throughput, it is crucial that no database server is overloaded. In other words, we need to place the table clusters such that we minimize the load of the most loaded server. This process is done in two steps. First, we evaluate the load imposed on each of the identified clusters for a representative workload. Second, we identify the placement that will create the best repartition of load across the servers.

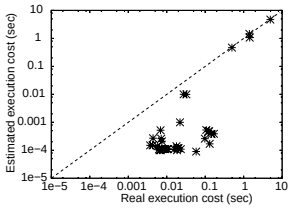
Estimation of load on table clusters

The load that each table cluster will impose on the server(s) where it is hosted depends on three factors: (i) the classification as belonging to a read or UDI template: read queries can be executed on one server, while UDI queries must be applied on all servers holding the corresponding cluster; (ii) the occurrence frequency of the template in the expected workload; and (iii) the computational complexity of executing the query on a given database server. Classifying queries as read or UDI can be done by simple query analysis. Similarly, the occurrence frequency of templates can be derived from observation of an existing workload. However, estimating the load that each query imposes on the database where it is run requires careful attention.

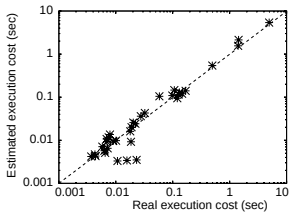
Mature database systems such as MySQL and PostgreSQL make their own estimations of the internal execution of different queries as part of their query optimization procedure. These execution time estimations are made available, for example using PostgreSQL's EXPLAIN <query> and EXPLAIN ANALYZE <query> commands. However, these estimations take only the actual execution time into



(a) EXPLAIN.



(b) EXPLAIN ANALYZE.



(c) Measurement under low load.

Figure 5.3: Accuracy of different methods for query cost estimation.

account, and ignore other factors such as the connection overhead. Another possible method consists of simply measuring the response time of each query template in a live system. The advantage of this method is that it measures the end-to-end response time of the database tier and includes connection overhead. Note that query execution times should be measured under low load to avoid polluting measurements with load-related overheads such as the queuing latency [Urgaonkar et al., 2005].

Figure 5.3 shows the accuracy of the three cost estimation techniques applied to the query templates from the RUBBoS benchmark. In each graph we estimated the cost of each query template and compared it with the actual execution time under high load. A perfect estimator would produce points located on the $y = x$ diagonal line. Clearly, the estimations produced by the database query optimizers are not as accurate as actual measurements made under low loads. Hence, in GlobeTP, we use the last method.

Estimation of load on database servers

In a replicated database, read queries are executed at one database node, whereas UDI queries are executed at all nodes that hold the data modified by the UDI query.

```
1 Distribute clusters uniformly across server nodes;
2 S = set of server nodes ;
3 while S ≠ ∅ do
      /*We want to minimize the maximum server load*/
4     while (max(estimated server load) is decreased) do
5         N = Most loaded server in S ;
6         foreach Cluster C placed in N do
7             Try to reduce N's load by migrating or replicating C to other
              servers;
          end
      end
8     S = S - (the most loaded server in S);
  end
```

Algorithm 2: Pseudocode of the cluster placement algorithm.

To determine the load that each database server incurs for a given table placement and a given query workload, we must distinguish the two types of queries.

Each UDI query in the studied workload will result in applying the associated execution cost to each of the database servers holding the corresponding table(s). On the other hand, each read query will create execution cost on only one server; we count that, on average, each database server holding the corresponding cluster will incur the execution cost of the query, divided by the number of replicas.

This analysis allows us to roughly compute the execution cost that each database server will incur for a given table placement and a given query workload. To maximize the system throughput, it is essential that no database server is overloaded. We therefore aim at balancing the load such that the cost of the most loaded server is minimized.

5.4.3. Cluster placement

Finding the optimal table placement can be realized by iterating through all valid table placements, evaluating the respective cost of each database server under each placement, and selecting the best one. However, the computational complexity of this exhaustive search is $O(2^{N*T}/N!)$, where T is the number of table clusters to be placed and N is the number of nodes to place them on. This very high complexity makes it unpractical even for relatively small system sizes. We must therefore find a heuristic instead.

As shown in Algorithm 2, our heuristic starts with a very roughly balanced placement, and iteratively tries to improve it by applying simple transformations in table placement.

The first step (step 1) is to place clusters uniformly onto servers to create an initial configuration. The heuristic then iteratively attempts to improve the quality of the placement (steps 3-8). Since the goal is to find the placement where the maximum server load is minimized, we identify the most loaded server (step 5) and try to offload some of its clusters to other servers (steps 6 and 7). Two techniques can be used here (step 7): either migrating one of the clusters to another server (thereby offloading the server of the whole associated load), or replicating one of the clusters to another server (thereby offloading the server of part of the read query load). The algorithm evaluates all possible operations of this type, and checks if one of them improves the quality of the placement. This operation is repeated until no more improvement can be gained (step 4). The most loaded server, which cannot be offloaded any more without overloading another one is considered to be in its 'optimal' state and is removed from the working set of servers (step 8). The algorithm then tries to optimize the load of the second most loaded server, and so on until all servers have reached their 'optimal' state.

Even though there is no guarantee that this heuristic will find the optimal placement, in our experience it always identifies a reasonably good placement within seconds (whereas the full search algorithm would take days).

5.4.4. Query routing

Query routing is an important issue that affects the performance of replicated databases. Simple round-robin schemes are efficient only when all the incoming queries have similar cost. However, when applications tend to have queries with different execution costs, round-robin scheduling can lead to load skews across database servers, resulting in poor access latencies.

Query routing gains a higher significance in GlobeTP. In a partially replicated database system such as ours, queries can no longer be sent to arbitrary database nodes. The query router used in GlobeTP thus differs from the traditional query routers used in fully replicated databases in the following aspects. First, read queries can be scheduled only among a subset of database servers instead of all servers. Second, UDI queries must executed at all database servers that store the tables modified by the incoming UDI query.

The process of selecting a database server to route an incoming read query has considerable impact on the overall performance of the system. This is usually determined by the routing policy adopted by the replication system. In our work, we experimented with the following policies.

RR-QID: Round-robin per query ID

RR-QID is an extension to the round-robin policy that is suitable for partially replicated databases. In this policy, the query router maintains a separate queue for each query template identified by its query identifier, QID. Each queue is associated with the set of database servers that can serve the incoming queries of type QID. Subsequently, each incoming read query is scheduled among the candidate servers (associated with its queue) in a round-robin fashion.

Cost-based routing

The underlying idea behind the cost-based routing policy is to utilize the execution cost estimations to balance the load among database servers. To this end, upon arrival of an incoming query, the query router first estimates the current load of each database server. Subsequently, it schedules the incoming query to the least loaded database server (that also has the required set of tables).

To this end, the query router maintains a list of queries that have been dispatched to each database server and still awaiting response. This list contains the list of queries currently under (or awaiting) execution at a database server. Subsequently, the load of a database server is approximated as the sum of the estimated cost of these queries. Finally, the server with least cost is scheduled to execute the next incoming query.

We show the respective performance of these two routing policies in the next section.

5.5. PERFORMANCE EVALUATION

In this section, we compare the performance of GlobeTP with full database replication for two well-known Web application benchmarks: RUBBoS and TPC-W. We selected these two applications for their different data access characteristics. This allows us to study the behavior of our systems for different data access patterns. In addition to these experiments, we also evaluate the benefit of adding a database query caching layer to GlobeTP.

5.5.1. Experimental setup

The TPC-W application uses a database with seven tables, which are queried by 23 read and 7 UDI templates. In our experiments the database is initially filled with $288,000$ customer records. Other tables are scaled according to the TPC-W requirements. TPC-W defines three standard workload mixes that exercise differ-

ent parts of the system: 'browsing' generates 5% update interactions; 'shopping' generates 20% update interactions; and 'ordering' generates 50% update interactions. We use the 'shopping' mix, which results in a database workload containing 5.6% of UDI queries[1].

The RUBBoS application consists of a set of PHP scripts and a database containing five tables regarding users, stories, comments, submissions and moderator activities. The database is initially filled with 500,000 users, out of which 10% have moderator privileges, and 200,000 comments. The size of the database is approximately 1.5 GB. The application defines 36 read and 8 UDI query templates. In our experiments, we used the default user workload which generates 0.76% of UDI queries.

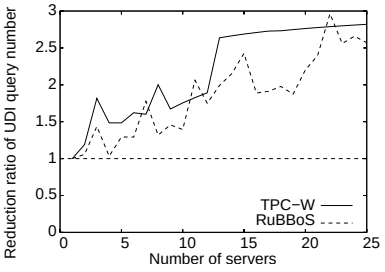
The client workload for both applications is generated by Emulated Browsers (EBs). The run-time behavior of an EB models a single active client session. Starting from the home page of the site, each EB uses a Customer Behavior Graph Model (a Markov chain with Web pages acting as nodes and navigation action probabilities as edges) to navigate among Web pages and perform a sequence of Web interactions. The behavior model also incorporates a think time parameter that controls the amount of time an EB waits between receiving a response and issuing the next request, thereby modeling a human user more accurately. We set the average think time to 6 seconds.

To generate flexible yet reproducible workloads, we run each benchmark under relatively low load (i.e., with 30 to 100 EBs) multiple times and collect the corresponding database query logs. Query logs from different runs can then be merged to generate higher load scenarios. For instance, to evaluate the performance of our system for 600 EBs, we merge the query logs from six different runs with 100 EBs, and stream the result to the query router. This allows us to study the performance of the database tier alone independently of other tiers.

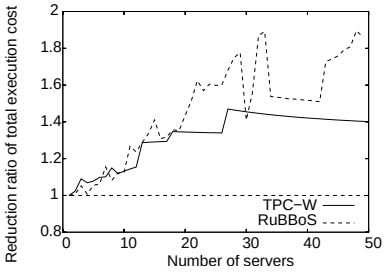
The query router is implemented as a stand-alone server written in Java. It maintains a pool of connections to each database server and schedules each incoming query based on the adopted routing policy. Database servers run PostgreSQL version 8.1.3. Both full and partial database replication are performed at the query router level as described in Section 5.3.2.

All our experiments are performed on a Linux-based server cluster. Each server is configured with dual-processor Pentium III 900 MHz CPU, 2 GB of memory and a 120 GB IDE hard disk. These servers are connected to each other with a gigabit LAN, so the network latency between the servers is negligible.

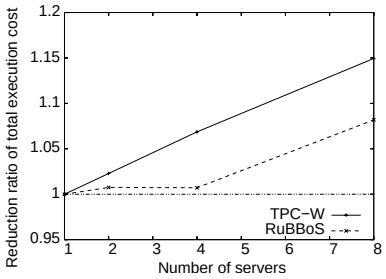
[1]Note that one must distinguish the update interactions from the UDI queries. Update interactions are user-generated HTTP requests that lead to at least one UDI query, plus any number of read queries. Since GlobeTP only operates at the database query level, the proportion of update interactions is irrelevant here.



(a) Reduction of the number of UDI queries.



(b) Reduction of estimated execution cost per server.



(c) Reduction of estimated execution cost per server (zoom).

Figure 5.4: Potential Reduction of UDI queries.

5.5.2. Potential reductions of UDI queries

One important goal of GlobeTP is to reduce the replication degree of individual database tables to reduce the number of UDI queries to be processed. However, the extent to which this is feasible greatly depends on the query templates issued by the application and the workload distribution.

Figure 5.4(a) shows to which extent table-level partial replication allows to reduce the number of UDI queries to be processed, expressed as the ratio of UDI queries to execute between full and partial replication. The higher the ratio, the greater the gain. Obviously, with just one server to host the database, partial and full replication are identical so the ratio is equal to 1. As the number of servers increases, partial replication allows to reduce the number of UDI queries by a ratio close to 3 for both applications.

To evaluate more accurately the potential load reduction that we can expect from partial replication, we should take into account the respective estimated costs

of different query templates, as well as the read queries from the workload. Figure 5.4(b) shows the reduction ratio of estimated costs imposed on each server, for different numbers of servers. As we can see, the reduction factor is much lower than when counting UDI queries alone. The reason is that both workloads are dominated by read queries, which are equally spread in full and partial replication.

The experiments described in the remaining of this chapter are based on configurations using up to 8 database servers. Figure 5.4(c) shows the respective potential of partial replication for both benchmarks under these conditions. TPC-W shows a relatively good potential, up to 15% reduction in workload per server. On the other hand, RUBBoS has a lower potential. This is mainly due to the fact that RUBBoS generates very few UDI queries; reducing their number even further by ways of partial replication can therefore have only a limited impact.

Note that other workloads may show somewhat different behavior. For example, RUBBoS defines a workload where search queries are disabled. Since searches are implemented as very expensive read queries, removing them from the workload mechanically improves the cost ratio of UDI queries and thereby the gains to be expected from GlobeTP. Conversely, we cannot exclude that other Web applications may dictate to keep all database tables together, making our form of partial replication equivalent to full replication. For these, GlobeTP will not provide any improvement unless the application itself is updated (see Section 5.6.1).

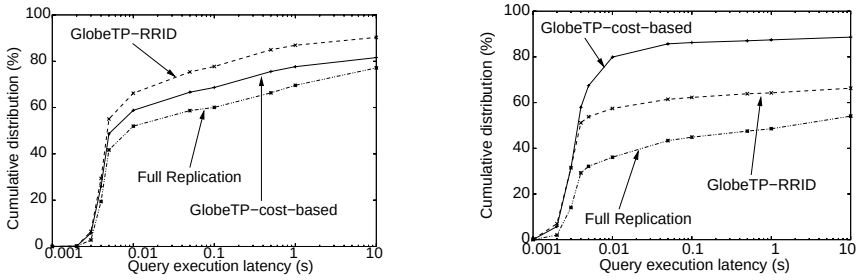
Here, we focus on standard benchmarks which offer real yet limited potential for use in our system. However, as we will see in the following sections, even the relatively modest reductions in estimated costs shown here allow for significant gains in execution latency and in total system throughput.

5.5.3. Partial replication and template-aware query routing

To illustrate the benefits of GlobeTP, we measured the query execution latencies using different configurations. For each of the two benchmarks, we compared the performance of full replication, GlobeTP using RR-QID query routing, and GlobeTP using cost-based query routing. In all cases we used 4 database servers and one query router. We selected a load of 900 EBs for TPC-W and 330 EBs for RUBBoS, so that the tested configurations would be significantly loaded.

Figure 5.5 shows the cumulative latency distributions from both sets of experiments. As one can see, in both cases GlobeTP processes queries with a much lower latency than full replication. For example, in RUBBOS GlobeTP processes 40% more queries than full replication within 10 ms. In TPC-W, GlobeTP processes 20% more queries within 10 ms than full replication.

In TPC-W, the RR-QID query routing policy delivers better performance than its cost-based counterpart. This can be explained by the fact that in TPC-W the



(a) TPC-W, 900 EBs. **(b)** RUBBoS, 330 EBs.

Figure 5.5: Query latency distributions using 4 servers.

costs of different query templates are relatively similar. The unavoidable inaccuracy of our cost estimations therefore generates unbalanced load across servers, which leads to sub-optimal performance. On the other hand, RR-QID is very effective at balancing the load when queries have similar cost.

In RUBBoS, GlobeTP combined with cost-based routing outperforms both other configurations. In this case, the costs of different queries vary by three orders of magnitude (as shown in Figure 5.3(c)). In this case, cost-based routing works well because even relatively coarse-grained estimations of the cost of each query helps avoiding issuing simple queries to already overloaded servers.

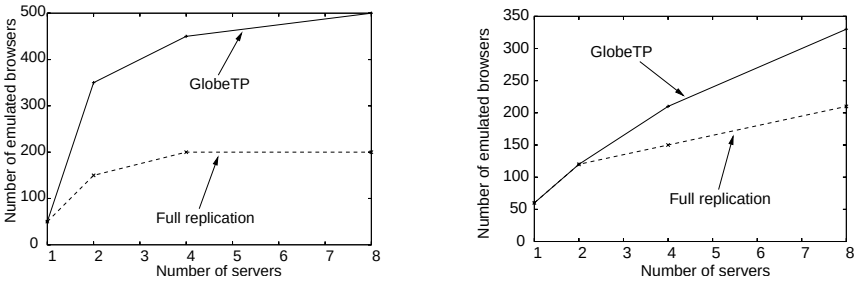
In the following experiments we restrict ourselves to the most effective routing policy for each application. We therefore use RR-QID for measurements of TPC-W, and cost-based routing for RUBBoS.

One should note that GlobeTP has greater effect on the latency in the case of RUBBoS than for TPC-W. This may seem contradictory with results from the previous section. However, the difference in latency characteristics is due to the difference in query execution costs. Moreover, as we will see in the next section, the throughput improvements that GlobeTP provides are significantly greater for TPC-W than RUBBoS.

5.5.4. Achievable throughput

To evaluate the scalability of our approach, we measured the maximum sustainable throughput of different approaches when using identical hardware resources. We first set a performance target in terms of query execution latency: in our experiments we aim at processing at least 90% of database queries within 100 ms. Note that this performance target is quite challenging, as several query templates have execution times greater than 100 ms, even under low loads (see Figure 5.3(c)).

We then exercise system configurations with full and partial replication, and



(a) TPC-W. **(b)** RUBBoS.

Figure 5.6: Maximum achievable throughputs with 90% of queries processed within 100ms.

increase the workload by steps of 50 EBs for TPC-W and 30 EBs for RUBBoS. For each configuration we record the maximum number of EBs that each configuration can serve while respecting the latency target. The results are shown in Figure 5.6.

In TPC-W, one server alone can sustain up to 50 EBs. As we increase the number of database servers, partial replication performs significantly better than full replication. In particular, the maximum throughput of the fully replicated system does not improve with more than four servers. This corresponds to the point when the treatment of UDI queries alone saturates each server. This can be explained by the fact that the execution time of a UDI query is typically an order of magnitude higher than that of a simple read query. On the other hand, GlobeTP can sustain up to 150% higher throughput while using identical hardware resources. Unlike full replication, it is capable of exploiting 8 servers to sustain higher throughput than when using only 4. This is due to the fact that each server has less UDI queries to process, and thereby experiences lower load and better execution latency.

In RUBBoS, GlobeTP again performs better than full replication, yet with a lower difference. With 4 and 8 servers, GlobeTP sustains 120 more EBs than full replication, which accounts for 57% of throughput improvement. Given that RUBBoS generates very few UDI queries, little improvement can be gained by further reducing their number with partial replication. In this case, the major reason for throughput improvement is the cost-aware query routing policy which takes the relative costs of different queries into account to better balance the load between servers.

Table 5.1: Maximum throughput of different configurations.

	TPC-W	RUBBoS
Full replication (4 servers)	200 EBs	150 EBs
GlobeTP (4 servers)	450 EBs	210 EBs
GlobeTP (4 servers) + 1 cache	600 EBs	330 EBs

5.5.5. Effect of query caching

As noted in Section 5.3.2, GlobeTP can easily be coupled with a database query caching system as most query caching systems rely on the same assumption as GlobeTP regarding the explicit definition of query templates. However, GlobeTP focuses on improving the throughput of the application's origin database, while query caching systems aim at reducing the individual query execution latencies. We therefore consider that both types of system complement each other very well. As a side effect, a query caching system can also improve the system throughput, as it prevents a number of read queries from being issued to the origin database.

In our experiments, we use our own in-house query caching solution, GlobeCBC. In our experiments, we limited the cache size to approximately 5% of the size of the database itself.

Table 5.1 shows the effect of adding a single cache server in front of the query router when using four database servers. In TPC-W, the cache had a hit rate of 18%. This relatively modest hit rate is due to the fact that the standard TPC-W workload has very low query locality compared to real e-commerce sites [Arlitt et al., 2001]. However, even in this case the system throughput is increased by 33%, from 450 to 600 EBs.

Unlike TPC-W, the RUBBoS workload has quite high database query locality. In this case the query cache delivers 48% hit ratio, which effectively increases the throughput by 57%, from 210 to 330 EBs. This result is quite remarkable considering that search queries, which are by far the most expensive ones in the workload, are based on random strings and are therefore always passed to the origin database for execution.

5.6. DISCUSSION

5.6.1. Potential of query rewriting

The above results demonstrate that relatively simple techniques allow to significantly improve the throughput of standard benchmarks, without requiring any modification to applications themselves. However, we believe that increased throughput can also be gained from simple changes of the application implementation.

The main limitation of the approach of table-granularity partial replication comes from database queries that span multiple tables. Such queries oblige the table placement algorithm to place all relevant tables together on at least one server, which in turn increases the replication degree and reduces the maximum throughput. Of course, queries spanning multiple tables are occasionally indispensable to the application. But we have observed from the TPC-W and RUBBoS benchmarks that many such queries can easily be rewritten as a sequence of simpler queries spanning one table each.

One simple example is the following query from TPC-W, which aims at obtaining all contact information about a given customer:

```
SELECT c_id, c_passwd, c_name, c_addr_id, co_id, [...] FROM tpcw_customer
JOIN tpcw_address on addr_id=c_addr_id JOIN tpcw_country on addr_co_id=co_id
WHERE c_uname=?
```

This query spans three tables. However the tables tpcw_address and tpcw_country are used here only to convert ZIP and country codes into their corresponding full-text descriptions. It is then trivial to rewrite the application to first issue a query to table tpcw_customer alone, then two more to convert the address and country codes separately.

We found such unnecessarily complex queries to be very frequent in the applications that we studied. Rewriting them into multiple simpler queries can only reduce the constraints put on the table placements, and therefore result in higher throughput.

5.6.2. Fault-tolerance

Although the main focus of GlobeTP is not replication for availability, one cannot ignore this issue. With increased number of server machines involved in a given application, the probability that one of them fails grows quickly. However, the most general form of fault-tolerance for this kind of system cannot be realized, as providing both consistency and availability in the presence of network partitions is impossible [Brewer, 2000; Gilbert and Lynch, 2002]. On the other hand, if we ignore the possibility of network partitions and restrict ourselves to server failures, then the problem has an elegant solution.

To guarantee that the partially replicated database remains able to serve all the expected query templates, it is essential that each query template be available at one or more servers. Therefore, to tolerate the failure of at most N servers one only has to make sure that each query template is placed on at least $N+1$ servers. This requires database replication algorithms suitable for fault-tolerance, which is a well-understood problem.

Failure of query router can result in losing queries that are awaiting execution and queries which have been successfully executed but whose response have not been sent back to the clients yet. Replication and recovery of the query router is an open issue and requires further study.

Starting from a configuration designed for throughput only, planning for fault-tolerance can be done in two different ways. First, one may keep the number of servers unchanged but artificially increase the replication degree of table clusters across the existing machines. However, this will likely degrade system throughput as more UDI queries must be processed per server. Alternatively, one may provision for additional servers, and adjust table placement to keep the worst-case throughput constant. As long as not too many servers fail, this configuration will exceed its throughput requirements, which may have the desirable side-effect of protecting the Web site to a certain extent against unexpected variations in load.

5.7. CONCLUSION

In this chapter we have presented GlobeTP, a database replication system that employs partial replication to optimize the system throughput. GlobeTP relies on the fact that the query workload of Web applications is composed of a restricted number of query templates, which allows us to determine efficient data placements. In addition, the identification of query templates allows for efficient query routing algorithms that take the respective query execution costs to better balance the query load. In our experiments, these two techniques allow to increase the system throughput by 57% to 150% compared to full replication, while using identical hardware configuration.

A natural extension of GlobeTP is to combine it with a database query caching system such as GlobeCBC, as both systems rely on the same definition of query templates. These systems complement each other very well: query caching improves the execution latency in a wide-area setting by filtering out many read queries, while GlobeTP shows its best potential for improving throughput under workloads that contain many UDI queries. In our experiments, the addition of a single query cache allows to improve the achievable throughput by approximately 30% to 60%.

CHAPTER 6

SLA-driven Resource Provisioning of Multi-tier Internet Applications

6.1. INTRODUCTION

Modern Web systems have evolved from simple monolithic systems to complex multi-tiered architectures. Web sites such as Amazon.com, yahoo.com and ebay.com often use such systems to generate content customized for each user. For instance, the Web page generated in response to each client request to Amazon.com is not generated by a single application but by hundreds of smaller Web applications operating simultaneously [Vogels, 2006]. Such elementary software applications, designed to be composed with each other are commonly referred to as *services*.

As shown in Figure 6.1, a service generates a response to each request by executing some application-specific business logic, which in turn typically issues queries to its data store and/or requests to other services. A service is exposed through well-defined client interfaces accessible over the network. Examples of services include those for processing orders and maintaining shopping carts. In such environments, providing low response latency is often considered a crucial business requirement and forms an integral component of customer service-level agreements (SLAs) [Shneiderman, 1984; Vogels, 2006].

In the previous chapters, we presented different database caching and replication techniques that aim to improve the application performance by alleviating the database bottleneck. Similarly, as we discussed in Chapter 2, various techniques have been proposed in the literature that replicate the database, application or Web servers to ensure that a service can meet its target SLA [Amiri et al., 2003a;

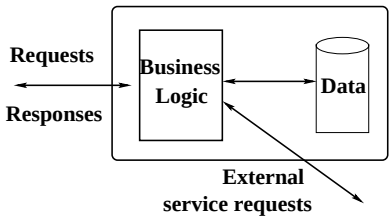


Figure 6.1: Simplified Application Model of an Internet Service

Bornhövd et al., 2004; Chen et al., 2006; Doyle et al., 2003; Menasce, 2003; Plattner and Alonso, 2004; Rabinovich et al., 2003; Seltzsam et al., 2006]. Each of these techniques aims to optimize the performance of a single tier of a service, and can be of great value for achieving scalability. However, from the viewpoint of an administrator, the real issue is not optimizing the performance of a single tier, but hosting a given service such that its end-to-end performance meets the SLA.

In the most general case, effective hosting of an Internet service involves replicating its code to a number of application servers and its data to an array of data store machines. Furthermore, different caching layers such as service response caches and database caches can be deployed to improve performance. As we demonstrate in this chapter, the widely adopted technique of scaling each tier of a service individually can result in gross overprovisioning and even poor performance. Instead, effective hosting of a service requires careful deployment of several techniques at different tiers. The underlying challenge then is, given a service and its workload, to find the *best resource configuration* for hosting the service in order to meet its SLA. We define the best resource configuration as the configuration that meets the SLA with the smallest quantity of resources.

Efficient provisioning of a multi-tier service poses three important challenges. First, choosing the right set of techniques to host a service in a scalable manner is not trivial as different techniques are best suited for different kinds of services. The best resource configuration for a service is therefore dependent on its characteristics and its workload. For example, if requests to the service exhibit high temporal locality and generate very few data updates, caching service responses might be beneficial. On the other hand, if the bottleneck is the retrieval of the underlying data, then database caching or replication might be preferable depending on the temporal locality of database queries. Sometimes, a combination of these techniques might be required to meet a certain SLA. Second, choosing the best resource configuration as the workload changes is challenging. This is because changes in the request workload have different impact on the performance of different tiers. For example, changes in the temporal locality of the requests will affect the performance of caching tiers but not that of the business logic tier. Third, the benefits of adding a resource to different tiers vary, and are highly de-

pendent on the nature of the service. For example, a computationally intensive tier such as a business logic tier can benefit from a linear decrease in queueing latency when adding a server. On the other hand, caching tiers follow a law of diminishing returns, and the benefit of increasing the number of cache servers decreases after a certain threshold.

The motivation behind the work presented in this chapter is based on the following observation: since the SLA of a multi-tier service is based on its end-to-end performance, it is not obvious that one can simply optimize the performance of each tier independently. As we will show, due to the complex interaction between tiers, such an approach does not allow to explain the overall performance of the service effectively, and moreover can lead to poorly performing or over-provisioned resource configurations. Instead, we claim that effective hosting of services requires a good understanding of the dependency between different tiers, and requires to make provisioning decisions based on an end-to-end performance model.

In this chapter, we present a novel approach to resource provisioning of multi-tier Internet services. Provisioning decisions are made based on an analytical model of the end-to-end response time of a multi-tiered service. A key observation behind our solution is that resource provisioning algorithms based on queueing models alone are not sufficiently effective as they do not capture the temporal properties of the workload such as the cache locality and the update characteristics. The proposed approach addresses this limitation by employing a combination of queueing models and runtime cache simulations. We have implemented a prototype system based on our approach for provisioning database-driven Web services. The implementation continuously gathers the required performance metrics (such as execution times at each tier, the incoming request rate, and cache hit ratio) using sensor software plugged into the different tiers. These data are passed to the resource provisioning system to decide on the right resource configuration for a given service to ensure that it meets its SLA. We demonstrate using several industry standard benchmarks that our approach makes the correct choices for different types of services and workloads. In many cases, our approach reaches the required performance-related SLA with less servers than traditional resource provisioning techniques. Finally, we discuss the problem of availability-based resource provisioning and discuss how we can extend our approach to meet availability related SLAs.

The rest of the chapter is structured as follows. Section 6.2 presents the background and overview of techniques used in hosting multi-tiered Internet services. Sections 6.3 and 6.4 present our analytical model and resource provisioning approach respectively. Section 6.5 presents experiments that demonstrate the effectiveness of our approach. Section 6.6 discusses the issues that arise from our

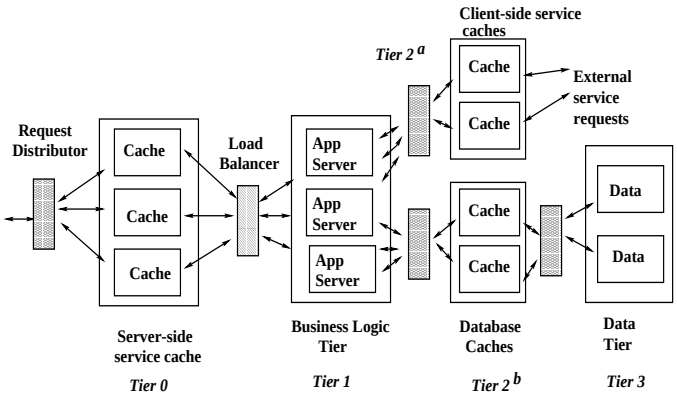


Figure 6.2: Generalized hosting architecture of a multi-tier service.

approach and the mechanisms to address them. Section 6.7 presents the related work and Section 6.8 concludes the chapter.

6.2. BACKGROUND

6.2.1. Infrastructure model

Typically, large-scale e-commerce enterprises deploy their software system across multiple data centers. A data center is built out of cluster(s) of servers connected to each other and to the Internet through a high-speed network. In this chapter, we assume that the resources allocated to a service are located within a single data center. Furthermore, we assume that each data center has a pre-allocated rescue pool of resources.

We assume that each service is assigned a performance and availability goal (usually referred to as an SLA). For sake of simplicity, we will initially restrict ourselves to performance-related SLAs. We define the SLA of a service such that its average response time should be within a $[LowRespTime, HighRespTime]$ interval. Availability related SLAs are discussed in Section 6.6.

We treat a single PC machine as the smallest unit of resource allocation. This approach is primarily motivated by the infrastructure model used by organizations such as Google and Amazon. Such organizations build their infrastructure out of inexpensive PCs instead of high-end multiprocessor server machines [Barroso et al., 2003; Vogels, 2006].[1] When a service hosted in a data center does not meet its SLA, one or more machines from the rescue pool are added to the service to

[1]For more description of the infrastructure model of Google, also see: http://www.nytimes.com/2006/06/14/technology/14search.html

ensure that it meets its desired performance. However, a service should not use more resources from the pool than necessary as these resources are shared with other services hosted in the data center.

6.2.2. Generalized service hosting architecture

As shown in Figure 6.1, a service conceptually consists of business logic that generates responses to incoming requests. To do so, it may issue (zero or more) queries to a service-specific database and (zero or more) requests to other services. Usually, the business logic of a Web service is hosted by a Web application server (e.g., IBM's Websphere[2] or JBoSS[3]). The data tier can use a relational DBMS (e.g., IBM's DB2[4], or MySQL[5]) or an object store such as [Ghemawat et al., 2003; Kubiatowicz et al., 2000]. In our work, we focus on relational DBMS-driven services. Moreover, we assume that services do not share their database but interact with each other only through their service interfaces.

The objective of our work is to determine the resource configuration for a service such that it can meet its SLA with the minimum number of resources. To do so, we need to identify the relevant tiers and their interrelationships. Instead of contrasting different caching and replication techniques, we claim that they are in fact complementary as they aim at improving the performance of different tiers. We define a generalized service hosting architecture encompassing several techniques that can be applied to improve the performance of a given service in addition to the functional tiers of business logic and data (see Figure 6.2). Any resource configuration where each tier of a service is provisioned with one or more servers (zero or more for non-functional tiers) will lead to a workable system. The problem of choosing the best resource configuration can now be translated to choosing one such workable resource configuration that will ensure that the service meets its SLA with minimum number of resources. We briefly explain the techniques used in our generalized service hosting architecture.

In *Server-side service-response caching* (done at tier 0), the system stores the response to each service request (identified by the requested method name and parameters), so that future identical requests are not forwarded to the business logic tier. The cached responses are often formed as XML messages if the service is implemented as a Web service. This technique can improve the throughput of the service as a cache hit in this tier reduces the load on the business logic tier (and other subsequent tiers). However, this technique is beneficial only if the service requests exhibit high temporal locality. Note that this is functionally similar to

[2] http://www.ibm.com/software/websphere

[3] http://www.jboss.org

[4] http://www.ibm.com/db2

[5] http://www.mysql.com

HTML fragment caching done for Web applications that deliver HTML pages. We discuss the mechanisms used to maintain the consistency of caches below.

Business-logic replication (done at tier 1) is a technique used when the computational load forms a bottleneck [Rabinovich et al., 2003; Seltzsam et al., 2006]. Usually, replicating a service's business logic translates to running multiple instances of the application across multiple machines. Examples of computationally intensive business logic include the page generation logic of an e-commerce site (that combines the responses from multiple services using XSL transformation to generate an HTML page) and Grid services.

Client-side service-response caching (done at tier 2^a) is used to cache the responses of the requests made by the business logic to the external services. As the external services can reside at other data centers, deploying a client-side service-response cache can be beneficial as it alleviates the wide-area network bottleneck. Intuitively, client-side response caches are useful only if there is high network latency between the client and the service or if the external service does not meet its SLA.

Database caching (done at tier 2^b) is a technique used to reduce the load on the data tier (see Chapters 2 and 3). In our prototype, we use our in-house database caching solution presented in Chapter 3, GlobeCBC. However, the approach and results presented in this chapter also apply to any other database caching system.

Database replication (done at tier 3) is another widely adopted technique to alleviate the database bottleneck. As discussed in Chapter 2, data are replicated usually to achieve better throughput and/or higher availability [Chen et al., 2006; Kemme and Alonso, 1998; Plattner and Alonso, 2004; Sivasubramanian et al., 2005]. If we restrict ourselves to performance considerations, then data replication is more beneficial than database caching only if the database queries exhibit low temporal locality [Sivasubramanian et al., 2006b]. On the other hand, if the underlying database receives many updates, then the benefit of database replication reduces due to the overhead of consistency maintenance.

6.2.3. Request distribution

In our service hosting architecture, each tier of a service can be provisioned with one or more servers (zero or more for caching tiers). This allows for increased service capacity but requires the use of load balancers to route the requests among the servers in a given tier. Usually, this task can be performed by hardware load balancers such as CISCO GSLBs[6], which are designed to uniformly distribute requests across a cluster of servers. However, balancing the request load across a distributed set of cache servers requires more sophisticated algorithms. A simple

[6] http://www.cisco.com/en/US/products/hw/contnetw/ps813/index.html

(weighted) round-robin request distribution technique can lead to a very poor hit rate as the request distribution is not based on where the requested object might be cached. We use consistent hashing to distribute requests across a distributed set of cache servers [Karger et al., 1999]. This design avoids redundant storage of objects across multiple cache servers. This means that adding cache servers will increase the number of unique cached objects, and can potentially improve the hit ratio. Another important feature of consistent hashing is that relatively few objects in the cluster become misplaced after the addition/removal of cache servers.

6.2.4. Cache consistency

Caching service responses (at tier 0 and tier 2^a) and database query responses (at tier 2^b) introduces a problem of consistency maintenance. A cached response at any of these tiers might become inconsistent when the underlying database of the service gets updated. In our system, we expect the developer to specify a priori which query template conflicts with which update template. As described in Chapter 3, these template definitions are used by our database caching system (GlobeCBC) to invalidate the cached query responses when the underlying database get updated. We use similar template-based invalidation technique for service-response caching.

6.3. MODELING END-TO-END SERVICE LATENCY

To choose the best resource configuration for a given service and its workload, we must be able to estimate the end-to-end latency of the service for any valid resource configuration. We first present a multi-queue-based analytical model for estimating the end-to-end latency of a multi-tiered Internet service. We then describe how an operational service can be fit to the model for estimating its latency under different resource configurations.

6.3.1. Analytical model

In general, multi-tiered systems can be modeled by open Jackson queueing networks [Jackson, 1957]. To illustrate this model, consider a system with k tiers. Let us assume that external requests arrive according to a Poisson process with a mean arrival rate γ_i to tier i. This assumption is usually true for arrivals from a large number of independent sources and has been shown to be realistic for e-commerce Web sites [Villela et al., 2004]. The requests at tier i receive service with an exponentially distributed duration with mean E_i. When a request completes service at tier i, it is routed to tier j with probability r_{ij} (independent of the

system state). There is a probability r_{i0} that the customer will leave the system upon completion of the service. In this model, there is no limit on the queueing capacity at each tier, thus a request is never blocked.

Jackson found the solution to the steady-state probabilities of the number of requests at each tier which is popularly known as a product form. Let λ_i be the total mean flow arrival into tier i (external requests and internally routed requests). Then, we have

$$\lambda_i = \gamma_i + \sum_{j=1}^{k} r_{ji}\lambda_j. \tag{6.1}$$

We define ρ_i to be $\lambda_i.E_i$. Let N_i be the random variable for the number of requests at tier i in steady state. Jackson showed that

$$\mathbb{P}(N_1 = n_1, \ldots, N_k = n_k) = (1-\rho_1)\rho_1^{n_1} \cdots (1-\rho_k)\rho_k^{n_k}.$$

This result states that the network *acts as if* each node could be viewed as an independent tier modeled by an M/M/1 queue with parameters λ_i and E_i. In fact, the network does not decompose into individual queues with Poisson arrivals, prohibiting the derivation of performance measures such as higher moments and variances. However, averages can be easily obtained for individual tiers, since the expected response time R_i of tier i is given by $R_i = \rho_i/[\lambda_i(1-\rho_i)]$. The expected total response time of a customer depends highly on the routing matrix $(r_{ij})_{ij}$. Let $z = (z_1, \ldots, z_m)$ be a path through the network, then the expected response time along this path is given by $\sum_{i=1}^{m} R_{z_i}$. By taking the expectation over all paths, the overall expected response time can be obtained. Naturally, for specific routing matrices, such as systems aligned in a serial order, this is straightforward.

The multi-tier systems considered in this chapter differ from the standard Jackson network in two aspects. First, the tiers should be modeled by processor-sharing systems instead of strict waiting queues. Second, the tiers have caches in front of them. However, [Baskett et al., 1975] tell us that the tiers can also be replaced by processor-sharing nodes with general service times. The given formulas and results still hold. Caching tiers also naturally fit into this model. For example, let us consider a scenario where a tier $i-1$ makes a request to a caching tier i to check if the response is already cached. If the response is found in the cache, then it is returned immediately else the request is forwarded to tier $i+1$ which generates the response. If p_i denotes the cache hit ratio for tier i, then the routing probabilities between the tiers can be formulated as $r_{(i-1)i} = p_i$ and $r_{(i-1)(i+1)} = 1-p_i$.

From the above results, we conclude that the expected end-to-end response time of our system can be obtained by taking the expectation of the response times taken by a request along its path. As shown in Figure 6.2(b), each incoming request is received by the first tier which in turns serves the request locally or

triggers calls to other tiers. Consider a tier T_i that receives a request that can be serviced locally with a probability p_i or can trigger multiple requests to more than one tier. Let K_i be the set of tiers that T_i calls for servicing its incoming requests, i.e., T_j ε K_i if T_j is called by T_i. For example, in Figure 6.2, T_1 makes requests to T_{2^a} and T_{2^b}, so $K_1 = \{T_{2^a}, T_{2^b}\}$. Let $N_{i,j}$ denote the average number of requests sent to T_j by T_i for serving a single incoming request to T_i, i.e., $N_{i,j} = r_{ij} * \lambda_i/\gamma_i$. For example, if a single request to the business logic tier (T_1) results in 1.5 queries to the data cache tier (T_{2^b}), then $N_{1,2^b} = 1.5$. The average response time, R_i, to service a request at T_i is given by:

$$R_i = Q_i + p_i * E_i + \sum_{j \varepsilon K_i} N_{i,j} * R_j \tag{6.2}$$

where Q_i is the average queueing latency experienced by a request at T_i before being serviced and E_i is the average time taken by tier T_i to execute the request (excluding the response times of other tiers). Equation 6.2 can capture the response times of tiers with different characteristics (e.g., caching or computational). For example, for a server-side caching tier (T_0), p_0 denotes the average cache hit ratio, $N_{0,1} = 1 - p_0$ (each request to the cache goes to T_1 only if it is a cache miss) and $K_0 = \{T_1\}$ (as all outgoing requests of T_0 are always sent to T_1). In the business logic tier, $p_1 = 1$, as all services always have to do some business logic computation, and $K_2 = \{T_{2^a}, T_{2^b}\}$ as the business logic can make requests to the external service tier (T_{2^a}) and data tier (T_{2^b}).

We can then perceive a service as a 4-tiered system, whose end-to-end response time can be obtained from equation 6.2 as follows:

$$\begin{aligned} R_0 = Q_0 + p_0 * E_0 + (1 - p_0) * (E_1 + N_{1,2^a} * R_{2^a} \\ + N_{1,2^b} * R_{2^b}) \end{aligned} \tag{6.3}$$

where R_1, R_{2^a} and R_{2^b} are the average response time for the business logic tier, client-side service-caching and database-caching, tiers respectively. These response times can be derived in a similar way from equation 6.2. We will discuss our model to compute variances and percentiles of response times in Section 6.6.

6.3.2. Service characterization

In this model, a service is characterized by parameters p_i, E_i and $N_{i,j}$. To estimate the average response time of a given service, these parameters must be measured. Most of these values can be obtained by instrumenting the cache managers and application servers appropriately. For example, the execution time of caches can be obtained by instrumenting the cache manager so that the average latency to

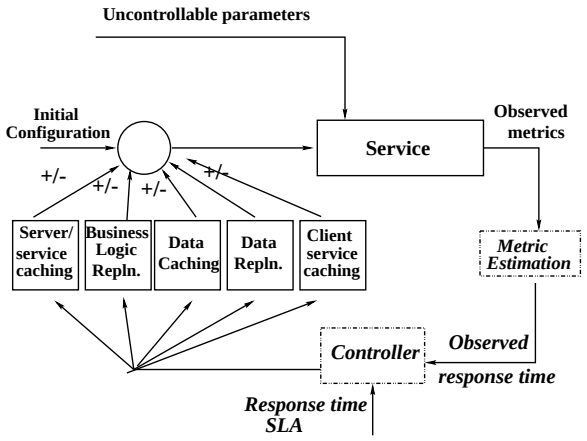


Figure 6.3: Logical Design of an Adaptive Hosting System for Internet Services

fetch an object from cache is logged. Note that all measurements of execution times should be realized during low loads to avoid measuring the queueing latency in addition to the service times [Urgaonkar et al., 2005]. Also, measuring the execution time of the business logic tier is less straightforward as instrumentation at this tier can obtain only the mean response time to service a request at the application server, i.e., R_1. However, R_1 is an aggregated value that encompasses E_1 and the response times of the subsequent tiers (see Equation 6.3). So, we must compute E_1 as $R_1 - R_{2^a} - R_{2^b}$. Note that the above model assumes that the calls made between tiers are synchronous. Modeling asynchronous calls is beyond the scope of this work.

Using this model only to estimate the current average response time of a service would be useless, as this value can also be measured directly. However, the model allows us to estimate the response time that the service would provide if its resource configuration (and therefore the parameters) was changed. In our experiments, we observed that the average response time estimates derived from our model have a high accuracy with an error margin less than 10%.

6.4. RESOURCE PROVISIONING

Once a service is running in our platform, we must monitor its end-to-end performance and, if necessary, decide on the necessary changes in its resource configuration. In the event of an SLA violation, the system may need to adapt the number of servers allocated to one or more tiers to ensure that its response time returns to and remains within its SLA. To this end, the system estimates the

response times that would result from different resource configurations using the model described in Section 6.3, and then choose the valid resource configuration that meets the SLA with the least resources.

In a sense, we can perceive the resource provisioning system as a typical continuous feedback system with appropriate sensors and controllers (shown in Figure 6.3). The system must detect changes in performance using sensors and trigger the controller to change the resource configuration (i.e., the number of servers provisioned for each tier) to bring the average response time back to the acceptable interval.

To do so, the controller needs to determine the improvement in response time that would result from adding a machine at each tier, and then choose the tier that gives the greatest improvement in end-to-end response time as the one to provision the resource. Adding an extra server to a tier T_i reduces the queueing latency Q_i as the tier has more processing capacity. Moreover, for caching tiers, this can improve the hit ratio (p_i) as the distributed cache has more storage capacity. To make good decisions, the controller therefore needs to determine the expected values of these parameters for each tier if the number of servers provisioned in T_i were changed.

6.4.1. Estimating Q_i

In general, the queueing latency, Q_i for a tier T_i with n servers can be computed using Little's law [Trivedi, 2002] as $Q_i = \lambda_i * E_i/n$, where λ_i is the arrival rate of requests to T_i and E_i is their mean execution time. Using this equation, the improvement in queueing latency of adding a new server can be easily estimated (assuming that the servers are homogeneous).

The above equation assumes that the request workload is uniformly shared among different servers in a tier. While this assumption is true for caching and business logic tiers, it is not the case for the databases (tier 3). In replicated databases, an update query must be processed by all database replicas, while read queries are processed by only one replica. Let w denote the fraction of update queries and $E_{dbwrites}$ be the average execution time for database updates. Similarly, let $E_{dbreads}$ denote the average read execution time. Then, the average queueing latency at a data tier with n database replicas is given as:

$$Q_{db} = \lambda_{db} * w * E_{dbwrite} + ((\lambda_{db} * (1-w) * E_{dbreads})/n) \tag{6.4}$$

where λ_{db} is the arrival rate of requests to the data tier. For estimating this value, we need the value of w, which requires explicit identification of update queries. As noted earlier, in our implementation we assume that each query is explicitly

marked and belongs to one of the read/write query templates. This allows us to obtain accurate measurements of *w* and average execution times.

6.4.2. Estimating improvement in cache hit ratio

For caching tiers, in addition to the execution time and queueing latency, the cache hit ratios have a major influence on the response time. Estimating the improvement in cache hit ratio when a new server is added is not trivial due to the non-linear behavior in hit ratio improvement. However, our approach relies on the ability to precisely estimate the gain in cache hit ratio that can be obtained by adding a new server to each distributed caching layer.

We estimate the possible gain in hit ratio due to the addition of a new server by running *virtual caches*. A virtual cache behaves like a real cache except that it stores only the metadata such as the list of objects in the cache, their sizes, and invalidation parameters. Objects themselves are not stored. By applying the same operations as a real cache, virtual caches can estimate the hit rate that would be offered by a real cache with the same configuration. Since a virtual cache stores only metadata, it requires relatively less memory. For example, to estimate the hit ratio of a cache that can hold millions of data items, the size of a virtual cache will be in the order of only a few megabytes. A virtual cache is usually run next to the real caches. When the resource configuration has no caches, virtual caches are run at the application server and database driver.

Let us assume the storage capacity of each cache server is M objects. In addition to running the real cache, each cache server runs a virtual cache with a capacity of $M+\Delta$ and logs its corresponding virtual cache hit ratio, number of cache hits and misses. The hit ratio of the virtual cache is what the server would have obtained if it had been given an extra Δ storage for caching. Consider a scenario where the caching tier runs N cache servers and Δ is set to M/N. Let $hits_{vc}$ denote the total number of cache hits at the virtual caches and *numreqs* be the total number of requests received by the caching tier. Then, $hits_{vc}/numreqs$ is the possible hit ratio the distributed cache would obtain when an extra M memory is added to it. This is equivalent to adding another server with memory M to the distributed cache. This estimation relies on the properties of consistent hashing: N servers each with capacity $M+M/N$ have the same hit ratio as $N+1$ servers with capacity M.

The hit ratio of the virtual cache is used by the controller to compute the gain in response time due to the addition of another server to the distributed cache (using equation 6.3). Similarly, the possible increase in response time due to removal of a server from the distributed cache can be estimated by maintaining another virtual cache in each cache server with a $M-\Delta$ storage capacity.

6.4.3. Decision process

In general, a change in resource configuration can be triggered by periodic evaluations or when the system observes an increase in end-to-end response time beyond the *HighRespTime* threshold set by the SLA. In such scenarios, the controller can use one or more servers from the rescue pool and add them to the service infrastructure to bring the response time back to the SLA interval. To do so, the controller must decide the best tier(s) to add the server(s). The controller obtains values of E_i, p_i, and $N_{i,j}$ for each tier from the metric measurement system. For caching tiers, it also obtains the estimated cache hit ratio for $M + \Delta$ memory. With these values, the controller computes R_0 values for the five different resource configurations that would be obtained when a server is added to one of the 5 tiers. Subsequently, it selects the configuration that offers the least end-to-end response time. This process is repeated until the response time falls within the acceptable interval or until the rescue pool is exhausted.

In multi-tiered systems, the software components at each tier have limits on the number of concurrent requests they can handle. For example, the Tomcat servlet engine uses a configurable parameter to limit the number of concurrent threads or processes that are spawned to service requests. This limit prevents the resident memory of the software from growing beyond the available RAM and prevents thrashing. Requests beyond this limit are dropped by the server. A provisioning system must therefore ensure that the servers in each tier of a chosen resource configuration will be handling requests well within their concurrency limit. This additional requirement can be handled by a simple refinement to the decision process. For a given resource configuration, in addition to estimating its response time the model can also estimate the concurrency degree at each tier, λ_i (using Equation 6.1). Subsequently, while selecting a resource configuration, we must ensure that the following two conditions are met: (i) the selected configuration meets the SLA with minimum number of resources and (ii) λ_i of each tier is less than its concurrency limit.

Continuous addition of servers without appropriate scaling down (over time) can lead to resource overprovisioning, thereby increasing operational costs of the system. To avoid this, the controller must periodically check if the observed response time is lower than the *LowRespTime* threshold. If so, the service is probably overprovisioned. The controller can then use the same technique to release the least useful resources, provided it does not lead to an SLA violation.

6.4.4. Prototype implementation

We implemented a prototype resource-provisioning system for evaluation purposes. The prototype is targeted for provisioning Java-based Web services that

transmit SOAP messages over HTTP. The prototype is based on Apache Jakarta Tomcat 4.1.31 running the Axis 1.5 SOAP engine, and MySQL MaxDB 5.0.19. In addition to these, we implemented our own service caches (server side and client side) and database cache in Java.

A service cache is essentially a simple server that receives service requests in XML over HTTP. Each request is assigned a unique identifier, *requestID*, which is a concatenation of the method name and its parameters. For example, if a service *getStockQuote*() is invoked to find the quote of a stock *id*1234 then its *requestID* value is set to *getStockQuote*(*id*1234). Each request carries zero or more *invalidationID* fields which identify the service requests whose invocation would invalidate the cached response of the request. For instance, in the above example if the service *getStockQuote*() conflicts with *updateStockQuote*(), then its *invalidationID* must be set to *updateStockQuote*(*id*1234).

We modified the Axis call object to add the fields *requestID* and *invalidationID* to each SOAP request. In our prototype, we require the application developer to explicitly set the *requestID* and *invalidationID* parameters for each request. This allows the cache servers to maintain the consistency of the cached service responses easily. We believe that such a requirement is not unrealistic, as usually application developers are well aware of the application's consistency semantics.

For database caching (T_{2^b}), we used GlobeCBC (described in Chapter 3). Database replication is implemented at the middleware level. We did not use the standard MySQL database cluster solution as it would require us to modify their code base to add the appropriate instrumentations for metric collection. Instead, database updates were serialized and scheduled from a single server. An update is applied only if it is successful at all replicas, else it is rolled back. Database reads are scheduled to replicas in a round-robin fashion.

In addition to the implementation of caching tiers, we instrumented the Jakarta Tomcat server and the caching tiers to log their execution times. These data are periodically collected by a monitor program and fed to a machine that runs the controller. The controller collects data from different tiers and constantly checks if the service's SLA is violated. If so, it decides on and triggers a change to the resource configuration based on the most recent data fed from different tiers.

As depicted in Figure 6.2, load balancers are responsible for distributing the load among application servers uniformly. In our experiment, we did not use any hardware load balancers but rather built a software-level request dispatcher that dispatches requests among application servers using a randomized round-robin algorithm. This allowed us to plug this component into the caches and service clients. The hash function and the membership table that describes the portion of object space covered by each cache server is essential for routing requests to them. In our experiments, this information was replicated at all cache clients. To notify

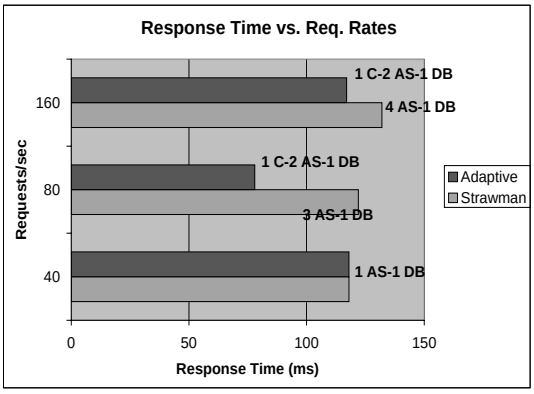


Figure 6.4: Performance of the Page Generator Service

other tiers regarding the addition/removal of servers in a tier, each tier implements an abstract method *changeResCfg(RoutingInfo)* and exposes it as a Web service. The controller invokes this method to notify the servers in a tier when the resource configuration of the tiers called by them is changed.

6.5. PERFORMANCE EVALUATION

In this section, we determine the effectiveness of our resource-provisioning approach in identifying the right configuration for different kinds of services and workloads. To this end, we evaluate the performance of our prototype for four different services: a page generator service, a customer personalization service, the TPC-App and RUBBoS benchmarks. These services are chosen to represent applications with different characteristics: a compute-intensive service, a data-intensive service, a business-to-business application encompassing several business interactions and a bulletin-board Web application. All the experiments were performed on dual processor Pentium III 900Mhz machines with 2GB memory running the Linux 2.4 kernel.

We compare the performance of our approach (which we call *adaptive*) against a *strawman* approach. The strawman approach adds or removes a server to the tier which is observed to have the highest bottleneck (measured in terms of the average CPU utilization of each server). This approach does not examine the temporal locality of request patterns. It is straightforward, and also widely used as it allows to provision resources independently across each tier [Rabinovich et al., 2003; Seltzsam et al., 2006].

6.5.1. Page generator service

A single request to an e-commerce Web site (like Amazon.com) often results in hundreds of service calls whose responses are aggregated by a front-end service to deliver the final HTML page to the clients [Vogels, 2006]. In this experiment, we compared the two resource-provisioning approaches for such a page generator service. We implemented a page generator service that, upon receipt of one client request, sends out requests to many other services and aggregates their responses. Emulating a page generator service of the scale of a large Web site requires us to run hundreds of other services. To reduce the experimentation complexity, we devised the page generator service as follows: Each request to the service triggers requests to two types of services. The first one is a customer detail service which returns customer information (first name, last name) based on a client identifier contained in the request. The second type of service is called a simple service, which is implemented as a TCP/IP socket server and returns the same XML response to all requests. By returning the same XML response to all requests, the simple service avoids the overhead of XML processing and hence does not become a bottleneck. In our experiments, the page generator service invokes one request to a customer detail service and separate requests to five different instances of the simple service. It then aggregates all the XML responses using XSL transformation, which is a CPU intensive operation. The customer identifiers in the request were generated according to a Zipf distribution with $\alpha = 0.4$ from a sample of 1 million customer identifiers. Similar values were observed on a measurement study profiling the workload of an e-commerce Web site [Arlitt et al., 2001]. The SLA of the service is arbitrarily set to $[50, 300]$ms.

In our experiments, we studied the response times for the resource configurations recommended by both provisioning systems for different request rates. The resulting configurations and their response times are shown in Figure 6.4. For all request rates the adaptive resource provisioning yields lower or equal response time than the strawman approach. In particular, for 160 req/sec, the adaptive provisioning is able to deliver better performance than the strawman approach while using less servers. This is primarily due to the fact that our approach accounts for and analyzes the temporal locality of the request workload. This enables the system to recommend adding a server-side service cache server rather than adding another application server. In our experiments, the hit ratio of the service cache is around 23% which significantly relieves the other tiers. In our experiments, we observed that the difference between the estimated response time derived from the model and the observed values were less than 7% of the latter.

It must be pointed out that since the primary differences between different customer responses arise from the customer service, in theory one could also install caches at tier T_{2^a} with excellent hit rates. However, as confirmed by the model,

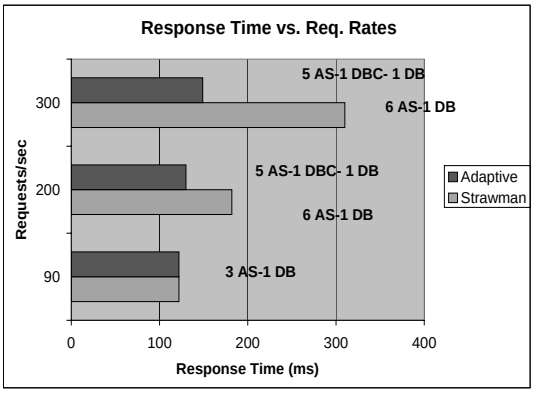


Figure 6.5: Performance for the promotional service

running cache servers at tier T_0 is better as the benefit of a cache hit is higher. On the other hand, the strawman approach observes that the application servers do the most load intensive task and addresses the bottleneck by adding more instances of the application server. This will only reduce the queueing latency by reducing the load imparted on each server. On the other hand, by introducing caching servers the adaptive provisioning also avoids execution latency (in addition to the queueing latency) incurred at subsequent tiers.

6.5.2. Promotional service

The second service we experimented with is a *promotional service* modeled after the "*Recommended for you*" service in Amazon.com. This service recommends products based on the user's previous activity. It maintains two database tables: (i) the item table contains details about each product (title, author, price, stock) and also identifiers of other items related to it; (ii) the customer table contains information regarding customers and the list of previous items bought by each customer. The business logic of this service executes two steps to generate a response. First, it queries the customer table to find the list of product identifiers to which a customer has shown prior interest. Second, it queries the item table to generate the list of items related to these products and their information. This service is mostly data intensive.

In our experiments, the item table was populated with 20 million randomly generated database records and the customer table was populated with 10 million records. We expect that such volumes of data are representative of a real-world application. The related items field of each record in the item table were populated randomly. The popularity of items among customers follows a Zipf distribution with $\alpha = 1$ (as shown in the analysis of a major e-commerce site [Arlitt et al., 2001]). For each type of query made by the business logic, the appropriate query

indices are created, which is a usual performance optimization technique used by enterprise database administrators. The client identifiers for each request were chosen randomly. The SLA of the service is set to [50,200]ms.[7]

Figure 6.5 shows the performance of both provisioning approaches for different request rates. For all request rates, the adaptive approach is able to obtain lower or equal latencies using the same number of servers. When the request rate for the service is increased from 90 req/sec to 200, the strawman approach simply looks at the tier with the highest load (which is the application server as it incurs XML parsing overhead) and adds 3 more servers. However, the bottleneck quickly turns out to be the database and hence the service fails to meet its SLA for a load of 300 requests/sec. In contrast, the adaptive approach collects information from different tiers regarding their execution times and hit ratios. In particular, the virtual database cache detected a high temporal locality among database queries and predicted a cache hit ratio of 62.5% with addition of one database cache server. On the other hand, the hit ratio of the virtual cache at tier T_0 was under 1% as the generated XML responses are different for each client. Using these values, the adaptive controller derived a configuration that reduces both the bottleneck at the business logic tier (by creating more instances of the application server) and at the database tier (by creating an instance of the database cache server). In this experiment, the error margin in mean response time estimations derived by the model were 3%, 6% and 9.5% for 90, 200 and 300 req/sec load respectively.

With these two experiments, we can draw the (preliminary) conclusion that for efficient resource provisioning we must: (i) optimize the end-to-end latency of the complete system and not just alleviate the bottleneck at individual tiers; and (ii) take into account the temporal properties of the request workload to derive accurate estimations of the impact of provisioning additional resources for different caching tiers. Furthermore, we can also see that the model has a high accuracy in deriving its response time estimations thereby allowing us to derive the right resource configuration for different workloads. In our subsequent experiments, we conduct an in-depth study of the performance of our resource provisioning system alone.

6.5.3. TPC-App: SOA benchmark

TPC-App is the latest Web services benchmark from the Transactions Processing Council.[8] It models an online shopping site like Amazon.com and performs business functionalities such as providing product details, order management and

[7] Once again, the SLA values are chosen arbitrarily. The SLA values only impact the decision regarding when to make a change in resource configuration and do not affect the performance of a running service.

[8] http://www.tpc.org/tpc_app/default.asp

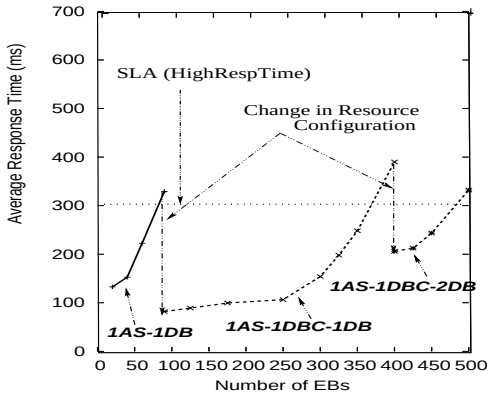


Figure 6.6: Performance for TPC-App benchmark

tracking, customer management, and shipment tracking and management. In contrast to its predecessor TPC-W[9], the interactions between the application and its clients are done through SOAP-based Web service interactions. The benchmark mandates the application to expose the following service interactions: *change item, product detail, order status, new products, change payment method, new customer, and create order*. The benchmark requires the clients to be implemented as Emulated Browsers (EBs) and to add some think time between subsequent requests. In our experiments, we used the mix workload recommended by the benchmark under which the workload received by each service interaction is split as follows: *New Customer* (1%), *Change Payment* (5%), *Create Order* (50%), *Order Status* (5%), *New Products* (7%), *Product Detail* (30%), *Change Item* (2%). The application uses 8 database tables that respectively store information on authors, customers, addresses, country, items, orders, orderlines, and stock. Even though the benchmark defines multiple service interactions they all share the same database, thereby effectively making it a single service (according to our application model definition).

We built our experiments upon the open source implementation of TPC-App.[10] However, this OSDL implementation implements the business logic as a Java-based servlet. The services were not implemented as Web services and the interaction between clients and services were not done using the SOAP protocol. To make this implementation conform to the TPC App benchmark, we ported it to the Jakarta Tomcat and Axis SOAP engine environment. We wrapped each service interaction into a Web service and the client EBs were modified appropriately. In our experiments, we filled the customer database with $288,000$ customer records. The other tables are scaled accordingly in conformance with the bench-

[9] http://www.tpc.org/tpcw/tpcw_ex.asp

[10] http://www.sourceforge.net/projects/osdldbt

mark regulations. The EBs issue requests to the application according to the mix workload with an average sleep time of 5 seconds. Contrary to the benchmark requirements, however, we do not guarantee ACID consistency as the database caches and service caches perform asynchronous invalidations.

We measured the response time of different resource configurations recommended by our system for different number of EBs. The SLA of the application is arbitrarily set to $[50, 300]$ ms as no standard values are recommended in the benchmark specifications. Changes in resource configuration are triggered whenever the running average of the response time fails to meet the SLA. Again, in this experiment, the model does a reasonable job in predicting the response time and the error margin is less than 10%. As can be seen in Figure 6.6, the resource configuration of the benchmark varies as the load increases. During low loads, a resource configuration with a single application server and one DBMS is sufficient for the system to meet its SLA. However, as expected, the response time increases with increase in client load. When the number of EBs grows beyond 90, the application fails to meet its SLA. Under such loads, the provisioning system diagnoses the underlying DBMS to be the bottleneck. Indeed, offline analysis confirms this observation as each service invocation to TPC-App leads to an average of 5.5 queries to the underlying database. Under this scenario, the virtual cache that is profiling the hit ratio of a database cache predicts a hit ratio of 52% which leads to a recommendation of adding a database cache to the resource configuration. As seen in Figure 6.6, this configuration keeps the end-to-end latency within its SLA until 300 EBs, beyond which the significant number of database query cache misses make the DBMS the bottleneck again. Under this scenario, the model recommends the addition of a DBMS instance. Note that in contrast to the addition of (service/database) caches or application servers, the dynamic addition of a DBMS instance is harder as it requires copying the entire database on the fly while processing updates without loss of consistency. In our experiments, we did not perform this online but the experiment setup was shutdown and a database replica was created and synchronized offline. We then measured the response time of this configuration. We discuss the issue of dynamic provisioning of servers in Section 6.6.

6.5.4. RUBBoS benchmark

Finally, we studied the resource configurations and response times obtained by our provisioning approach for the RUBBoS benchmark, a bulletin board application that models slashdot.org. In contrast to all previous studied applications, the RUBBoS application models a Web site and not a Web service, i.e., it does not use SOAP to interact with its clients. The main reason for studying this benchmark is to demonstrate the applicability of our approach to a wide variety of multi-tiered

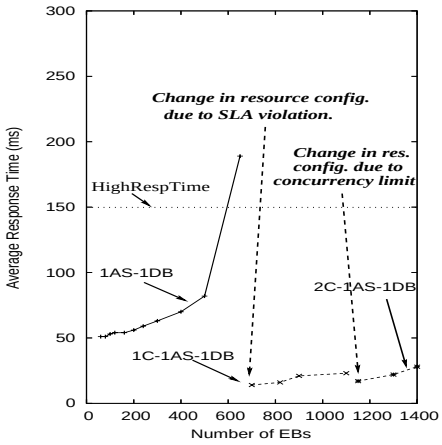


Figure 6.7: Resource configurations and response times of RUBBoS benchmark for different loads.

applications. RUBBoS's database consists of five tables, storing information regarding users, stories, comments, submissions, and moderator activities. We filled the database with information on 500,000 users and 200,000 comments. Our experiments were performed with the browse only workload mix. The average think time between subsequent requests of a client session was set to 6 seconds. For our experiments, we chose the open source implementation of these benchmarks[11]. The SLA of the Web site was arbitrarily fixed to [10,150] ms.

Figure 6.7 presents the response times of the benchmark that runs a single application server and a backend database. As expected, the response time increases as the number of client EBs increases and when we reach beyond 700 EBs, the SLA is violated. The virtual cache (corresponding to T_0) running at the application server predicts a hit ratio of 91%. The high hit ratio can be explained by the fact that for a news Web site like Slashdot, many readers exhibit the same browsing patterns such as viewing the abstracts of the top stories of the day, details of the top stories. This observation leads the controller to recommend the addition of a server at T_0, i.e., an HTML cache. The addition of a HTML cache makes the latency drop by almost an order of magnitude, which is consistent with a hit ratio of 90%. This also allows the resource configuration to sustain a much higher load.

Interestingly, even though the response time is well within its SLA, the T_0's cache concurrency limit is reached when the number of EBs is increased to 1100 EBs. Under this load, the T_0 cache starts dropping many incoming requests. At this point, the system recommends addition of another T_0 cache to keep the request rate well within the concurrency limits of the cache.

[11] http://jmob.objectweb.org/rubbos.html

6.6. DISCUSSION

6.6.1. Performance of reactive provisioning

A key issue in the design of any dynamic resource provisioning system is how reactive the provisioning system is bringing the system behavior back to its SLA when the workload changes. The time to get the system back to its SLA depends on three factors: (i) the time taken to make a decision on the best resource configuration, (ii) the time taken to make the resource changes, and (iii) the time taken for the new resource configuration to warm up. In our system, the controller is constantly fed with performance metrics so making a decision on the changes in resource configuration is purely a local constant time computation. The time incurred due to the second issue heavily depends on the availability of resources and if the software systems in different tiers are "designed" to be dynamically provisioned. If enough resources are available in the rescue pool, then dynamic provisioning of software systems in each tier is the main issue. By the virtue of virtualization, we believe caching systems and application servers can be installed on a new server instantly. For instance, dynamically creating instances of a Java-based application is now relatively well understood and even industry-standard application servers such as WebSphere allow the dynamic creation of application instances [Naik et al., 2004]. However, dynamic creation of a replica at the database tier is more difficult, especially for RDBMSs. This is because creating a replica of a relational database while the service is running requires copying the entire database without affecting the database performance and ensuring that all updates made on the original database during the process of copying are executed consistently on the final replica. These issues are beginning to be addressed (see e.g., [Chen et al., 2006]). However, current techniques are by no means highly reactive. We acknowledge that this remains very much an open research issue.

Once the resource configuration has been put in place, the new configuration does incur some "warm-up" overhead. This can be due to the initialization of connection pools, thread pools or cache warmups. While connection and thread pool initialization are not specific to the running service, cache warmup times heavily depend on the service and workload. Cache warmup time is the time incurred before the hit ratio of a cache stabilizes. It of course heavily influences the time incurred by the system to fall back to its SLA as the virtual cache predicts the hit ratio of a "warmed-up" cache. Cache warmup times heavily depend on the workload's request rate and temporal locality. To give an example of a typical cache warmup time, we plotted the running average of the response time for the TPC-App benchmark before and after the addition of a database cache (for the first change in resource configuration shown in Figure 6.6). As shown in Figure 6.8, the response time of the system decreases immediately after adding a database

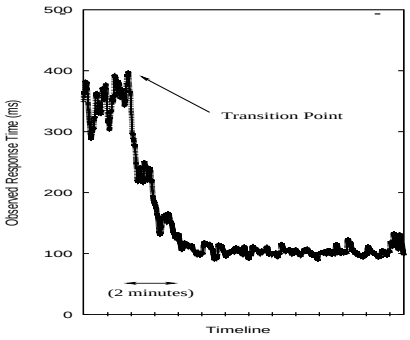
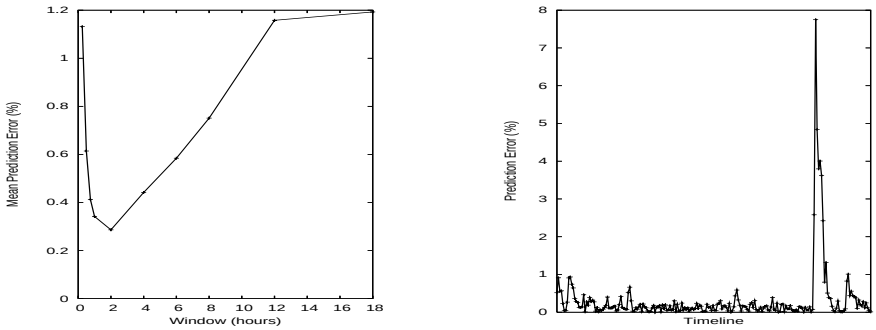


Figure 6.8: Observed response time during a change in the resource configuration: From a single application server and database to a configuration with an application server, database cache and a database.



(a) **(b)**

Figure 6.9: (a) Impact of window size on hit ratio prediction error and (b) Plot of the hit ratio prediction error of the fractals Web site during the flash crowds.

cache server. In our experiment, the time taken for the response time to stabilize was under 2 minutes.

We believe that virtualization allows us to dynamically provision different tiers with considerable ease (as observed in [Clark et al., 2005]). However, dynamically provisioning databases still remains a major obstacle. This leads us to conclude that, for the time being, the best option is to perform a pessimistic provisioning of the database tier.

6.6.2. Predictability of cache hit rates

A key building block of the provisioning system is the use of a virtual cache for predicting the effectiveness of caching techniques at different tiers. For accurate

predictions, the hit ratios must be measured only over a relatively large time window. A small window size will make the prediction mechanism susceptible to errors due to small load bursts and cache warmup effects. On the other hand, a large window size will make the system unreactive to detect changes in temporal locality.

To study the impact of window size on prediction accuracy, we examined the access logs of a Web site that experienced flash crowds due to its posting on the Google home page and subsequently on Slashdot the next day.[12] The trace contains all requests received by the server over a period of two months. In our first experiment, we measured the impact of the window size on the quality of the hit ratio prediction during the first month when there was no flash crowd. The cache size was set to 0.1% of the total number of unique Web objects. In our experiment, a prediction was made every sampling interval. The frequency of sampling was set to 10% of the window length. Figure 6.9(a) plots the impact of window sizes on the mean prediction error. The hit ratio prediction error is the absolute difference between the hit ratio predicted for a given time period and the actual value observed for the time period. The lower the value, the higher is the prediction accuracy. As seen, the optimal window length for these traces is around 2 hours, which gives the best prediction accuracy with a prediction error lower than 0.3%. An important point to be noted here is that a history window of 2 hours seems to be reactive enough to capture the "time-of-the-day" effects.

As a next step, we wanted to see the effect of flash crowds on a system that runs an optimal virtual cache predictor. To this end, we examined the requests during the flash crowd and plotted the hit ratio prediction error during this period. The size of the history window is set to 2 hours which was determined to be optimal during stable load scenarios. The results of our experiment are given in Figure 6.9(b). The prediction error turns out to be lower than 0.5% during stable loads and raises to 8% during flash crowds. Needless to say, this implies that predictions made during the flash-crowd event are inaccurate. The reason is that flash crowds for static Web sites are characterized by sudden and huge increase in request rate for a small number of Web objects. This leads to a sudden increase in hit ratio (a similar observation was made in [Jung et al., 2002b]). However, the sudden increase in hit ratio leads to inaccurate predictions. Note that flash crowds need not always lead to improved cache hit ratio. For instance, a flash crowd on a customer service might result in lower locality as the response generated for each customer will be different. As of now, we do not have any precise mechanisms to handle predictions during flash crowds. We speculate that adaptations to be performed during flash crowds might require mechanisms of different nature such as enhanced admission control, and reduced consistency requirements. This means

[12] http://local.wasp.uwa.edu.au/~pbourke/fractals/quatjulia/google.html

that the system would operate in an "emergency mode" during this period. It is to be expected that the system should not use the measurements collected during this mode to decide on its adaptations during normal operations.

6.6.3. Modeling variances and percentiles

In Section 6.3, we modeled the system as a Jackson queueing network. This model has the appealing feature that the network acts as if each tier can be viewed independently of the other tiers. This makes a mean-value analysis effective so that performance measures that deal with averages can be easily obtained. However, it does not allow us to derive other measures such as variance and percentiles. Under some reasonable assumptions, it is possible to obtain tight bounds for these performance measures. This is one of the primary focusses of the work presented in the next chapter.

6.6.4. Availability-based provisioning

The model so far assumes that servers do not fail. This is of course unrealistic as server failures lead to loss of processing capacity (thereby leading to high response times) and sometimes even system outages. In that case, it is imperative to base provisioning decisions not just on end-to-end response time but also on availability requirements. Note that, availability is a fundamentally different performance measure than the end-to-end latency. When the software system has a tree structure, such as depicted in Figure 6.2(b), the total system availability is dominated by the tier with the smallest availability measure. Hence, the availability measure turns out to be a local performance measure. Therefore, to achieve end-to-end availability of the system it is enough to optimize the availability of each tier independently. This leads us to conclude that to achieve both performance and availability SLAs, one can provision based on end-to-end performance and subsequently add redundancy at each tier to achieve a desired level of availability.

6.7. RELATED WORK

As discussed in Chapter 2, a vast number of solutions have been proposed for improving the performance of Web applications. Systems such as [Challenger et al., 2005] cache application responses in HTML or XML. These techniques (that correspond to T_0 in the generalized hosting architecture given in Figure 6.2(b)) improve the throughput of the service as a cache hit in this tier offloads the complete application from serving the request. Systems such as ACDN [Rabinovich

et al., 2003] and Autoglobe [Seltzsam et al., 2006] aim at scalable hosting of Web applications by replicating/migrating the application code across different servers. Database replication middlewares [Kemme and Alonso, 1998; Plattner and Alonso, 2004; Sivasubramanian et al., 2005] aim at alleviating the database bottleneck by replicating the entire database. Database caching middlewares [Amiri et al., 2003a; Bornhövd et al., 2004; Sivasubramanian et al., 2006b] aim to alleviate the database bottleneck by caching database query responses. All these techniques are studied independently and aim to address bottlenecks at different tiers of a service. As shown in our evaluations, for effective hosting of Internet services one should optimize the end-to-end performance of a tier instead of optimizing individual tiers.

Our problem is closely related to capacity provisioning and has been well studied in the context of single-tiered applications [Doyle et al., 2003; Menasce, 2003]. A simple transposition of these techniques to our problem is, however, not suitable as database, business logic, and service caches have very different characteristics. Hence, it is imperative to treat each individual tier as a separate entity. In a recent study [Urgaonkar et al., 2005], the problem of provisioning a 3-tier web site using multi-queueing models has been addressed. Unfortunately, the study is entirely based on multi-queueing models and therefore cannot take into account any caching techniques (such as client/server-side service caching or database caching). This is a very limiting restriction as caching is one of the widely used techniques used in boosting the performance of a service.

6.8. CONCLUSION

In this chapter, we presented a novel approach to resource provisioning of multi-tier Internet services. In contrast to previous works on resource provisioning, our approach selects the resource configuration based on its end-to-end performance instead of optimizing each tier individually. Our proposed approach employs a combination of queueing models and on-line cache simulations and is capable of analyzing the impact of temporal properties of the workload on the end-to-end response time. We demonstrated through extensive experimentations the effectiveness of our approach in achieving the best resource configuration for different applications and industry standard benchmarks. Compared to the strawman approach, our approach maintains the SLA of a service with less number of servers. Even though this work primarily focusses on maintaining average latencies in an environment with no server failures, we discussed how our model can be refined to take latency percentiles and server failures into account in the provisioning decision.

CHAPTER 7

Analysis of End-to-End Response Times of Multi-Tier Internet Services

7.1. INTRODUCTION

In the previous chapter, we demonstrated how we can do effective resource provisioning based on a simple analytical model. However, the aforedescribed model allows us to estimate only the mean end-to-end response time of an application and does not provide any bounds on its variability. In this chapter, we present an analytical model for multi-tiered software systems and derive exact and approximate expressions for the mean and the variance, respectively, of the end-to-end response times.

In contrast to most previous works on modeling Internet systems, we do not restrict ourselves to the mean value but also provide accurate approximations to its variance. Such approximations are very desirable as most e-commerce organizations measure the client experience based on variability in the response times in addition to the mean value [Vogels, 2006]. Hence, obtaining bounds on the end-to-end response times, even if approximate, is highly beneficial.

Deriving an analytical model for multi-tier Internet applications involves the following challenges. First, multi-tier applications often exhibit complex interactions between different tiers. A single request to the application can lead to multiple interactions between the business logic and database tiers. This creates a strong dependence which prohibits derivation of higher moments, such as the variance, whereas deriving the mean is still tractable. Second, the resource configuration of multi-tier systems is highly dynamic in nature. For instance, caching tiers can be added/removed based on the temporal properties of its workload. Un-

der such circumstances, the model should be flexible enough to accommodate the changes in the resource configuration of the application. Finally, many multi-tiered applications achieve scalability by employing caching techniques to alleviate the load at different tiers. Examples of such caching techniques include [Amiri et al., 2003a; Bornhövd et al., 2004; Olston et al., 2005; Li et al., 2003; Datta et al., 2002]. Any performance model should be able to incorporate these techniques such that performance measures are still accurate.

Contributions

We model a multi-tiered application as a system in which each request arrives at a single entry node which in turn issues requests to other nodes (see Figure 7.2). Each tier might have a cache in front of it which offloads its computational burden with a certain probability. Otherwise, the request is executed by the computational node. The request arrival distribution is assumed to be a Poisson process, which is usually true for arrivals from a large number of independent sources and shown to be realistic for many Internet systems [Villela et al., 2004]. The entry node can be equated to a business logic tier that receives the service requests, and the service nodes correspond to databases/other services that are queried upon for serving the request. Using this model, we derive expressions for the mean end-to-end response time and approximations to its variance.

The contributions of this chapter are threefold. First, we develop an analytical model for the end-to-end response times for multi-tiered Internet applications. Second, we derive exact and approximate expressions for the mean and the variance, respectively, of the end-to-end response times. We validate these expressions through simulation first. Subsequently, we validate the model using several industry-standard benchmarks (running on a Linux-based server cluster). Our experiments demonstrate that our model accurately predicts the mean response time and its variance for different application and resource configurations. Finally, we show how our model can be applied to several scenarios, such as resource provisioning, admission control, and SLA-based negotiation.

Our proposed model has several advantages:

- *Performance Prediction:* The model allows designers and administrators to predict the end-to-end performance of the system for a given hardware and software configuration under different load conditions.

- *Application Configuration Selection:* To ensure that an application can meet its performance targets, various techniques have been proposed that replicate or cache the database, application, or Web servers [Menasce, 2003; Doyle et al., 2003; Chen et al., 2006; Amiri et al., 2003a; Bornhövd et al., 2004; Plattner and Alonso, 2004; Rabinovich et al., 2003; Seltzsam et al.,

2006; Olston et al., 2005; Li et al., 2003; Datta et al., 2002]. Each of these techniques aims to optimize the performance of a single tier of a service, and can be of great value for achieving scalability. However, from the viewpoint of an administrator, the real issue is not optimizing the performance of a single tier, but hosting a given service such that its end-to-end performance meets the SLA. For effective selection of the right set of techniques to apply to an application, we need to identify its bottleneck tier(s). Again, an end-to-end analytical model can help in identifying such bottlenecks and allow us to tune the system performance accordingly.

- *Capacity Planning:* To ensure that a multi-tiered application meets its desired level of performance, each tier must be provisioned with enough hardware resources. Our model can enable the administrators to decide on the right number of servers to allocate to each tier of an application such that its end-to-end response time is within the bounds defined by the business requirements.

- *Request Policing:* An application needs to reject excess number of requests during overload situations to meet its performance targets. Our model enables the application to determine the point when and which requests to allow to enter the system.

The rest of the chapter is structured as follows. In Section 7.2 we present the related work. In Section 7.3 we present our analytical model. Based on this model, we derive the expressions for the mean response time and its variance in Section 7.4. We first validate the model through simulations in Section 7.5. Subsequently we demonstrate the accuracy of the model using two industry-standard benchmarks running on a Linux-based server cluster in Section 7.6. In Section 7.7 we discuss and show the benefits of the model relating to issues such as resource provisioning, service level agreement (SLA) negotiation, and admission control. Section 7.8 concludes the chapter.

7.2. RELATED WORK

7.2.1. Modeling Internet systems

Various research works in the past have studied the problem of modeling Internet systems. Typical works include those modeling Web servers, database servers, and application servers [Menasce, 2003; Doyle et al., 2003; Chen et al., 2006; Kamra et al., 2004]. For example, in [Menasce, 2003], the authors use a queueing model for predicting the performance of Web servers by explicitly modeling

the CPU, memory, and disk bandwidth in addition to using the distribution of file popularity. Bennani and Menasce [Bennani and Menasce, 2005] present an algorithm for allocating resources to a single-tiered application by using simple analytical models. Villela et al. [Villela et al., 2004] use an M/G/1/PS queueing model for business logic tiers and to provision the capacity of application servers. An G/G/1 based queueing model for modeling replicated Web servers is proposed in [Urgaonkar and Shenoy, 2005], which is to perform admission control during overload situations. In contrast to these queueing based approaches, a feedback control based model was proposed in [Abdelzaher et al., 2002]. In this work, the authors demonstrate that by determining the right handles for sensors and actuators, the Web servers can provide stable performance and be resilient to unpredictable traffic. A novel approach to modeling and predicting the performance of a database is proposed in [Chen et al., 2006]. In this work, the authors employ machine learning and use an K-nearest neighbor algorithm to predict the performance of database servers during different workloads. However, the algorithm requires substantial input during the training period to perform effective prediction. In [Seltzsam et al., 2006], the authors propose the use of fuzzy logic controller to change the number of instances of application code running in a system.

All the aforementioned research efforts have been applied only to single-tiered applications (Web servers, databases, or batch applications) and do not study complex multi-tiered applications which form the focus of this chapter. Some recent works have focused on modeling multi-tier systems. In [Kamra et al., 2004], the authors model a multi-tiered application as a single queue to predict the performance of a 3-tiered Web site. As mentioned, Urgaonkar et al. [Urgaonkar et al., 2005] model multi-tier applications as a network of queues and assume the request flows between queues to be independent. This assumption enables them to assume a product-form network so that they can apply mean value analysis (MVA) to obtain the mean response time to process a request in the system. Although this approach can be very effective, MVA approaches can be limiting in nature as they do not allow us to get variances which are also of crucial importance in large scale enterprises [Vogels, 2006].

7.2.2. Performance analysis

There are few works that study the performance of Internet systems in the context of multi-tier applications. Although the response time was made explicit for the first time in [Mei and Meeuwissen, 2006], much research on modeling response times has already been done for other systems. Results for the business logic, modeled as a processor sharing (PS) node, are given in [Coffman et al., 1970], where the Laplace-Stieltjes Transform (LST) is obtained for the M/M/1/PS node. In [Morrison, 1985] an integral representation for this distribution is derived, and

in [Ott, 1984] the distribution is derived for the more general M/G/1/PS system (in case a representation of the service times is given). The individual services, behind the business logic, are usually modeled as first-come-first-served (FCFS) queueing systems for which results are given in [Cooper, 1981].

The first important results for calculating response times in a queueing network are given in [Jackson, 1957], in which product-form networks are introduced. A multi-tier system modeled as a queueing network is of product-form when the following three conditions are met. First, the arrival process is a Poisson process and the arrival rate is independent of the number of requests in the network. Second, the duration of the services (behind the business logic) should be exponentially distributed when FCFS queues are used, but can have a general distribution in case of PS or infinite server queues. Moreover, the duration is not allowed to depend on the number of requests present at that service. Finally, the sequence in which the services are visited is not allowed to depend on the state of the system except for the state of the node at which the request resides. Multi-tier systems that satisfy these properties fall within the class of so-called Jackson networks and have nice properties.

In [Boxma and Daduna, 1990], the authors give an overview of results on response times in queueing networks. In particular, they give expressions for the LST of the joint probability distribution for nodes which are traversed by requests in a product-form network according to a pre-defined path. Response times in a two-node network with feedback (such as at the business logic) were studied in [Boxma et al., 2005]. The authors propose some solid approximations for the response times with iterative requests. They show that the approximations perform very well for the first moment of the response times. In [Mei et al., 2006], a single PS node is studied with several multi-server FCFS nodes. The authors derive exact results for the mean response time as well as estimates for the variance. The performance analysis in this chapter is an extension of their work.

7.3. END-TO-END ANALYTICAL MODEL

In this section, we develop a model for multi-tier Internet services in the context of a queueing-theoretical framework. For this purpose, consider a queueing network with $2(N+1)$ nodes as depicted in Figure 7.2. Requests that are initiated by an end-user arrive according to a Poisson process with rate λ to a dedicated entry level 0. The request may be cached at caching tier 0 with probability p_0. In that case, the request is served by caching tier 0 and the response is directly delivered to the end-user. However, with probability $1-p_0$, the request needs to be processed by service tier 0 and the other N levels in the queueing network, which

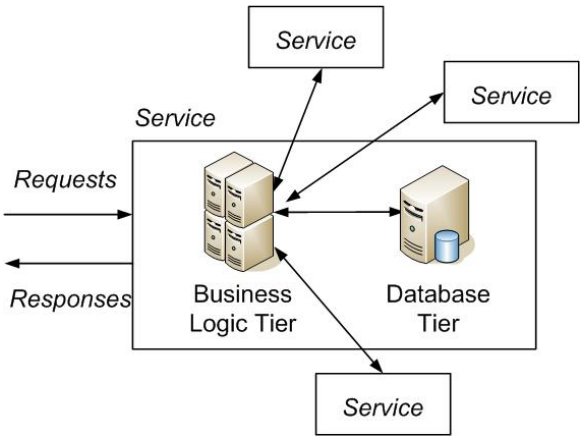


Figure 7.1: Application Model of a Multi-Tiered Internet Service.

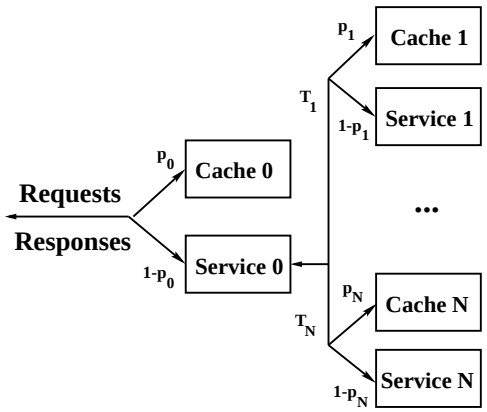


Figure 7.2: Analytical Model for Multi-Tiered Applications .

can be identified with the business logic and the existing basic services delivered by the service provider, respectively, as depicted in Figure 7.1.

As mentioned before, when a request is not cached at caching tier 0, it traverses the network by first visiting service tier 0. After service completion, the request is routed to each of the N levels in sequence. At level i, the request is served by caching tier i with probability p_i. When the request is not cached at the caching tier (which occurs with probability $1 - p_i$), the request is served by service tier i. After having received service at level i, the results are sent back for processing to service tier 0 (which takes place with the same parameters as upon first entry). We assume that a request that is sent from the business logic to level i generates T_i requests back and forth, where T_i is a non-negative discrete random variable. Note that this parameter allows us to accurately capture the typical be-

havior of Web applications. For instance, the business logic code of many Internet applications makes multiple queries to its underlying databases (or other services) to generate a single response. Thus, a request has fully completed its service at level i after T_i service completions. Finally, every request that is served by the service tier 0, passes this tier $1+T_1+\cdots+T_N$ times, and finally leaves the system after having visited all N levels.

We model each caching tier i by an infinite-server queue having general service times with an average of $\beta_{c,i}$ time units for $i=0,\ldots,N$. Each service tier i is modeled by a processing sharing queue and draws its service times from a general probability distribution with an average service time of $\beta_{s,i}$ time units for $i=0,\ldots,N$. The mean response time of the service is modeled as the sojourn time of the request in the system. Let $S_i^{(k)}$ be the sojourn time of the k-th visit to level i, and $M=\mathbb{E}T_1+\cdots+\mathbb{E}T_N$. Then, the expected sojourn time $\mathbb{E}S$ of an arbitrary request arriving to the system is given by

$$\mathbb{E}S=\mathbb{E}\left[\sum_{k=1}^{M+1}S_0^{(k)}+\sum_{i=1}^{N}\sum_{j=1}^{T_i}S_i^{(j)}\right].$$

Note that the system is modeled such that it satisfies the conditions of a product-form network. First, the arrivals occur according to a Poisson process with a rate that is not state dependent. This is not unrealistic in practice, since arrivals from a large number of independent sources do satisfy the properties of a Poisson process [Villela et al., 2004]. Second, the basic services are modeled by mixtures of processor sharing and infinite server queues with a general service distribution which does not depend on the number of requests at that node. Finally, since the sequence in which the service nodes are visited is fixed, and thus does not depend on the state of the system, the network is of product-form.

7.4. MEAN RESPONSE TIME AND ITS VARIANCE

In the previous section we have seen that the queueing network is of product-form. Consequently, when $L_{c,i}$ and $L_{s,i}$ denote the stationary number of requests at caching tier i and service tier i for $i=1,\ldots,N$, respectively, we have

$$\mathbb{P}(L_{c,i}=l_{c,i},L_{s,i}=l_{s,i};\,i=0,\ldots,N)=$$
$$\prod_{i=0}^{N}\mathbb{P}(L_{c,i}=l_{c,i})\prod_{i=0}^{N}\mathbb{P}(L_{s,i}=l_{s,i}),$$

with $l_{c,i}$ and $l_{s,i}=0,1,\ldots$ for $i=0,\ldots,N$. From this expression, the expected sojourn time at the entry node and the service nodes can be determined. First,

define the load on the entry nodes by $\rho_{c,0} = \lambda p_0 \beta_{c,0}$ for the caching tier and $\rho_{s,0} = (M+1)(1-p_0)\lambda\beta_{s,0}$ for the service tier. Similarly, the load for the basic services is given by $\rho_{c,i} = (1-p_0)p_i\lambda\beta_{c,i}$ and $\rho_{s,i} = (1-p_0)(1-p_i)\lambda\beta_{s,i}$ for $i = 1,\ldots,N$. Then, by Little's Law, the expected sojourn time $\mathbb{E}S_{c,i}$ at caching tier i is given by $\mathbb{E}S_{c,i} = \beta_{c,i}$, whereas the expected sojourn time $\mathbb{E}S_{s,i}$ at service tier i is given by $\mathbb{E}S_{s,i} = \beta_{s,i}/(1-\rho_{s,i})$ for $i = 0,\ldots,N$. Combining all the expressions for the expected sojourn time at each tier in the network, we derive that the expected response time is given by

$$\begin{aligned}
\mathbb{E}S &= \mathbb{E}\left[\sum_{k=1}^{M+1} S_0^{(k)} + \sum_{i=1}^{N}\sum_{j=1}^{T_i} S_i^{(j)}\right] = p_0\,\mathbb{E}S_{c,0} + \\
&\quad (1-p_0)\left[\mathbb{E}S_{s,0} + \sum_{i=1}^{N}\mathbb{E}T_i\left\{p_i\mathbb{E}S_{c,i} + (1-p_i)\mathbb{E}S_{s,i}\right\}\right] \\
&= p_0\beta_{c,0} + (1-p_0)\frac{(M+1)\beta_{s,0}}{1-\rho_{s,0}} + \\
&\quad (1-p_0)\sum_{i=1}^{N}\mathbb{E}T_i\left[p_i\beta_{c,i} + (1-p_i)\frac{\beta_{s,i}}{1-\rho_{s,i}}\right].
\end{aligned}$$

Let us now focus our attention to the variance of the response times. It is notoriously hard to obtain exact results for the variance. Therefore, we approximate the total sojourn time of a request at service tier 0 by the sum of $M+1$ independent identically distributed sojourn times. Moreover, we approximate the variance by imposing the assumption that the sojourn times at the entry node and the sojourn times at the service nodes are uncorrelated. In that case, we have

$$\begin{aligned}
\mathbb{V}\mathrm{ar}\,S &= \mathbb{V}\mathrm{ar}\left[\sum_{k=1}^{M+1} S_0^{(k)} + \sum_{i=1}^{N}\sum_{j=1}^{T_i} S_i^{(j)}\right] \\
&= \mathbb{V}\mathrm{ar}\left[\sum_{k=1}^{M+1} S_0^{(k)}\right] + \mathbb{V}\mathrm{ar}\left[\sum_{i=1}^{N}\sum_{j=1}^{T_i} S_i^{(j)}\right].
\end{aligned}$$

To approximate the variance of the sojourn times at tier 0, we use the linear interpolation of Van den Berg and Boxma in [Berg and Boxma, 1991] to obtain the second moment of the sojourn time of an M/G/1/PS node. We adapt the expression by considering the $M+1$ visits together as one visit with a service time that is a convolution of $M+1$ service times. Let $c_{s,i}$ denote the coefficient of variation of the service times at service tier i for $i = 0,\ldots,N$. Then, we have that the variance of the sojourn time $S_{s,0}$ at service tier 0 is approximately given by $\mathbb{V}\mathrm{ar}\,S_{s,0}(M+1)$,

with

$$\mathbb{V}\text{ar}\, S_{s,i}(K) = (K+1)c_{s,i}^2\left[1+\frac{2+\rho_{s,i}}{2-\rho_{s,i}}\right]\left[\frac{\beta_{s,i}}{1-\rho_{s,i}}\right]^2 - \left[\frac{(K+1)\beta_{s,i}}{1-\rho_{s,i}}\right]^2 + \left((K+1)^2-(K+1)c_{s,i}^2\right)\times \left[\frac{2\beta_{s,i}^2}{(1-\rho_{s,i})^2} - \frac{2\beta_{s,i}^2}{\rho_{s,i}^2(1-\rho_{s,i})}(e^{\rho_{s,i}}-1-\rho_{s,i})\right].$$

In principle, the same expression for the variance of service tiers i can be used with $M=0$, yielding the expression $\mathbb{V}\text{ar}\, S_{s,i}(1)$ for $i=1,\ldots,N$. Recall that a single request to tier 0 can lead to multiple interactions between tier 0 and tier i. This introduces additional variability W_i due to various reasons such as context switching, request and response processing. We estimate this variability generated by the random variable T_i by Wald's equation [Tijms, 1994] as follows.

$$\mathbb{V}\text{ar}\, W_i = \mathbb{V}\text{ar}\, T_i\Bigg\{\mathbb{V}\text{ar}\, S_{s,0}(\mathbb{E}T_i) + \mathbb{V}\text{ar}\, S_{s,i}(1) + (1-p_0)^2\times \left(\frac{(\mathbb{E}T_i+1)\beta_{s,0}}{1-\rho_{s,0}} + \mathbb{E}T_i\left[p_i\beta_{c,i}+(1-p_i)\frac{\beta_{s,i}}{1-\rho_{s,i}}\right]\right)^2\Bigg\}.$$

Let $\beta_{c,i}^{(2)}$ denote the second moment of the service times at caching tier i for $i=0,\ldots,N$. Then, the variance $\mathbb{V}\text{ar}\, S_{c,i}$ for caching tier i is given by

$$\mathbb{V}\text{ar}\, S_{c,i} = \beta_{c,i}^{(2)} - \beta_{c,i}^2.$$

Finally, by combining all the expressions for the variances of the sojourn times at each node in the network, we derive that the variance of the response time is given by

$$\mathbb{V}\text{ar}\, S \approx p_0^2\,\mathbb{V}\text{ar}\, S_{c,0} + (1-p_0)^2\left\{\mathbb{V}\text{ar}\, S_{s,0}(M+1) + \sum_{i=1}^{N}\left[p_i^2\,\mathbb{V}\text{ar}\, S_{c,i} + (1-p_i)^2\mathbb{V}\text{ar}\, S_{s,i}(1) + \mathbb{V}\text{ar}\, W_i\right]\right\}.$$

Note that the expressions exhibit a lot of structure and clearly show how each tier contributes to the mean and the variance. This makes the expressions highly flexible so that changes in the resource configuration of the application can be easily accommodated for. For instance, suppose that service tier i is replaced with an FCFS queueing system with c_i servers having exponentially distributed service times with mean $\beta_{s,i}$ (note that this retains the product form solution). Then, only

β_{ps}	c_{ps}^2	β_{fcfs}		$\mathbb{V}\text{ar}_s\, S$	$\mathbb{V}\text{ar}\, S$	$\Delta\mathbb{V}\text{ar}$ %
0.1	0	0.1	0.9	82.51	81.05	−1.33
0.3	0	0.8	0.5	85.89	86.81	1.06
0.1	0	0.5	0.3	1.25	1.22	−1.76
0.3	0	0.9	0.1	147.49	150.82	2.25
0.1	4	0.1	0.9	80.83	81.33	0.62
0.3	4	0.8	0.5	274.00	278.46	1.63
0.1	4	0.5	0.3	1.54	1.50	−2.68
0.3	4	0.9	0.1	331.30	342.47	3.37
0.1	16	0.1	0.9	81.55	82.16	0.75
0.3	16	0.8	0.5	831.15	853.41	2.68
0.1	16	0.5	0.3	2.37	2.33	−1.86
0.3	16	0.9	0.1	871.49	917.42	5.27

Table 7.1: Response time variances of a queueing network with general service times at the entry node and two asymmetrically loaded single-server service nodes.

$\mathbb{E}S_{s,i}$ and $\mathbb{V}\text{ar}\, S_{s,i}$ have to be changed. To this end, let $\rho_{s,i} = (1-p_0)p_i\lambda\beta_{s,i}/c_i$, and define the probability of delay π_i by

$$\pi_i = \frac{(c_i\rho_{s,i})^{c_i}}{c_i!}\left[(1-\rho_{s,i})\sum_{l=0}^{c_i-1}\frac{(c_i\rho_{s,i})^l}{l!} + \frac{(c_i\rho_{s,i})^{c_i}}{c_i!}\right]^{-1}.$$

Then, the mean sojourn time $\mathbb{E}S_{s,i}$ at service tier i is given by

$$\mathbb{E}S_{s,i} = \frac{\beta_{s,i}}{(1-\rho_{s,i})c_i}\,\pi_i + \beta_{s,i},$$

and the variance of the sojourn time $\mathbb{V}\text{ar}\, S_{s,i}$ at service tier i is approximated by

$$\mathbb{V}\text{ar}\, S_{s,i} \approx \frac{\pi_i(2-\pi_i)\beta_{s,i}^2}{c_{s,i}^2(1-\rho_{s,i})^2} + \beta_{s,i}^2.$$

7.5. VALIDATION WITH SIMULATIONS

In this section we assess the quality of the expressions of the mean response time and the variance that were derived in the previous section. We perform some numerical experiments to test validity of the expressions against a simulated system. In this case, the mean response time does not need to be validated, because the results are exact due to [Jackson, 1957]. Therefore, we can restrict our attention to validating the variance only.

We have performed extensive numerical experiments to check the accuracy of the variance approximation for many parameter combinations. This was achieved by varying the arrival rate, the service time distributions, the asymmetry in the loads of the nodes, and the number of servers at the service nodes. We calculated the relative error by $\Delta\mathbb{V}\text{ar}\ \% = 100\% \cdot \left(\mathbb{V}\text{ar}\ S - \mathbb{V}\text{ar}_s S\right)/\mathbb{V}\text{ar}_s S$, where $\mathbb{V}\text{ar}_s S$ is the variance based on the simulations.

We have considered many test cases. We started with a queueing network with exponential service times at the entry node and two service nodes at the backend. In the cases where the service nodes were equally loaded and asymmetrically loaded, we observed that the relative error was smaller than 3% and 6%, respectively. We also validated our approximation for a network with five single-server service nodes. The results demonstrate that the approximation is still accurate even for very highly loaded systems. Based on these results, we expect that the approximation will be accurate for an arbitrary number of service nodes. The reason is that cross-correlations between different nodes in the network disappear as the number of nodes increases. Since the cross-correlation terms have not been included in the approximation (because of our initial assumptions), we expect the approximation to have good performance in those cases as well.

Since the approximation for different configurations with single-server service nodes turned out to be good, we turned our attention to multi-server service nodes. We carried out the previous experiments with symmetric and asymmetric loads on the multi-server service nodes while keeping the service times at the entry nodes exponential. Both cases yielded relative errors smaller than 6%. Finally, we changed the service distribution at the entry node. Table 7.1 shows the results for a variety of parameters, where the coefficient of variation for the service times at the entry nodes is varied between 0 (deterministic), 4 and 16 (Gamma distribution). These results are extended in Table 7.2 with multi-server service nodes. If we look at the results, we see that the approximation is accurate in all cases. To conclude, the approximation covers a wide range of different configurations and is therefore reliable enough to obtain the variance of the response time.

7.6. VALIDATION WITH EXPERIMENTS

In the previous section, we demonstrated the accuracy of our mean and variance expressions using simulations. In this section, we validate our model with two well-known benchmarks: RUBBoS, a popular bulletin board Web application benchmark, and TPC-App, the latest benchmark from the Transactions Processing Council modeling service oriented multi-tiered systems. The choice of these two applications was motivated by their differences in their behavior.

β_{ps}	c_{ps}^2	β_{fcfs}		$\mathbb{V}\mathrm{ar}_s\, S$	$\mathbb{V}\mathrm{ar}\, S$	$\Delta\mathbb{V}\mathrm{ar}$ %
0.1	0	0.2	2.7	80.82	85.66	5.99
0.3	0	1.6	1.5	88.82	89.70	0.99
0.1	0	1.0	0.9	2.43	2.43	−0.02
0.3	0	1.8	0.3	149.31	152.38	2.05
0.1	4	0.2	2.7	88.50	85.94	−2.90
0.3	4	1.6	1.5	272.00	281.35	3.44
0.1	4	1.0	0.9	2.71	2.71	−0.23
0.3	4	1.8	0.3	330.14	344.03	4.21
0.1	16	0.2	2.7	89.60	86.77	−3.16
0.3	16	1.6	1.5	820.26	856.30	4.39
0.1	16	1.0	0.9	3.57	3.57	−0.79
0.3	16	1.8	0.3	920.45	918.98	−0.16

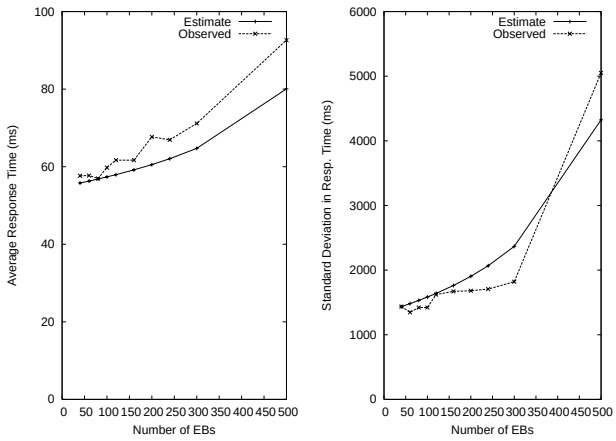
Table 7.2: Response time variances of a queueing network with general service times at the entry node and two asymmetrically loaded multi-server service nodes.

We validate our model by running these benchmarks on a Linux-based server cluster for the following resource configurations: (i) when the application is deployed with a single application server and a database server, (ii) when the application is deployed with a front-end cache server, an application server and a database server, and (iii) when the application is deployed with a single application server, a database cache server and a database server. We first present our experimental setup followed by the validation results.

7.6.1. Experimental setup

We hosted the business logic of these applications in the Apache Tomcat/Axis platform and used MySQL 3.23 for database servers. We ran our experiments on Pentium *III* machines with 900 Mhz CPU and 2 GB memory running a Linux 2.4 kernel. These servers belonged to the same cluster and network latency between the clusters was less than a millisecond. In our experiments, we varied the arrival rate and measured the mean end-to-end response and its variance and compared the measured latency with the values predicted by the model.

In our model, an application (or a service) is characterized by the parameters β_i, β_{T_i} and var_i, where $i = 0 \cdots N$. Therefore, to accurately estimate the mean and the variance of the response times for a given application, we first need to obtain these values. Most of these values are obtained by instrumenting the cache managers, application servers and database servers appropriately. For example, the execution time of caches can be obtained by instrumenting the cache manager so that the average latency to fetch an object from the cache is logged. Note that all measurements of execution times should be realized during low loads to avoid



(a) Mean Response Time **(b)** Standard Deviation

Figure 7.3: Comparison between the observed and the predicted values for the mean and the standard deviation of the response times for RUBBoS benchmark with a single application server and a single database server.

measuring the queueing latency in addition to the service times. This method has been successfully employed in similar problems [Urgaonkar et al., 2005].

As in the previous chapters, the client workload for both benchmarks is generated by Emulated Browsers (EBs). The mean think time between subsequent requests of a client session in an EB is set to 5 seconds.

7.6.2. RUBBoS: A bulletin board Web application

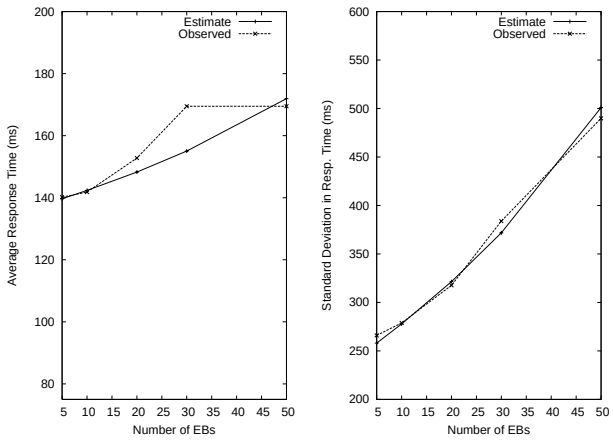
In our first set of experiments, we experimented with the RUBBoS benchmark that we have described in detail in the previous chapters. We experimented with a single application server (hosting the RUBBoS application code) and a single database server (storing application data). This corresponds to a system with two service tiers, which is a special instance of the model proposed in Section 7.3. The mean response time reduces to

$$\mathbb{E}S = \frac{(\mathbb{E}T_1 + 1)\beta_{s,0}}{1 - \rho_{s,0}} + \mathbb{E}T_1 \frac{\beta_{s,1}}{1 - \rho_{s,1}},$$

and the variance is approximated by

$$\mathbb{V}\text{ar}\, S \approx \mathbb{V}\text{ar}\, S_{s,0}(\mathbb{E}T_1 + 1) + \mathbb{V}\text{ar}\, S_{s,1}(1) + \mathbb{V}\text{ar}\, W_1.$$

The instrumentation collected during low load measured $\beta_{s,0} = 0.482$ ms, $\beta_{s,1} = 1.69$ ms, $\mathbb{V}\text{ar}\, S_{s,1} = 16.33$, $\mathbb{E}T_1 = 25.1$ and $\mathbb{V}\text{ar}\, T_1 = 412.5$. These values indicate



(a) Mean Response Time **(b)** Standard Deviation

Figure 7.4: Comparison between the observed and the predicted values for the mean and the standard deviation of the response times for TPC-App Benchmark with a single application server and a single database server.

that the mean database query latency is relatively low. Moreover, it also shows that the RUBBoS's application code makes a relatively large number of requests to the underlying database for generating a single response.

In our experiments, we varied the client load by varying the number of EBs and measured the mean response times and its variance. All experiments were run for a period of 2 hours with a warm up time of 15 minutes. We compared these values with the predicted values obtained from the model. The results are shown in Figure 7.3. As seen in the figure, the model does a good job in predicting not just the mean response time but also its standard deviation. These experiments show that the margin of error is less than 12% in most of the cases. We believe these results are commendable considering the model does a reasonably accurate prediction even for high loads (such as when the application is handling 500 concurrent sessions).

7.6.3. TPC-App: A service-oriented benchmark

In our next set of experiment, we used the TPC-App benchmark. For more details on TPC-App, we refer to Chapter 6. The resource configuration used in this experiment is similar to the configuration used in the previous section, resulting in the same expressions for the mean and the variance. Similar to the previous experiment, we measured the application's performance during low load to obtain

the relevant parameters for the model. The values obtained from our instrumentation were $\beta_{s,0} = 0.8556\,\text{ms}$, $\beta_{s,1} = 11.18\,\text{ms}$, $\mathbb{V}\text{ar}\, S_{s,1} = 10730$, $\mathbb{E}T_1 = 6.51$ and $\mathbb{V}\text{ar}\, T_1 = 27.2$. From these values, we can infer that the characteristics of this application is vastly different from RUBBoS. For instance, TPC-App's mean database query latency is significantly higher compared to that of RUBBoS. On the other hand, TPC-App's application code makes relatively less number of requests to the underlying database for generating a single response.

As in the previous experiment, we varied the client load by varying the number of EBs and measured the mean and standard deviation in response times for different loads. All experiments were run for a period of 1 hour with a warm up time of 15 minutes. We compared these values with the predicted values obtained from the model and the results are shown in Figure 7.4. As seen in the figure, the response time predictions of our model is highly accurate even for variances. In particular, the accuracy of variance predictions is less than 5% which we believe is quite commendable.

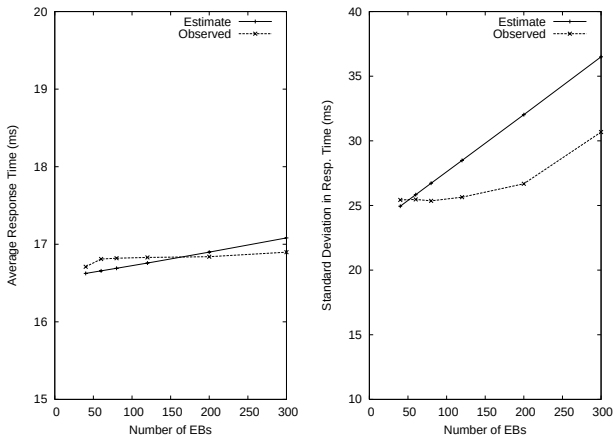
As can be seen, the model's prediction is significantly higher for TPC-App (error margin less than 5%) compared to RUBBoS (error margins usually less than 10%). We believe the difference in the accuracy is mainly due to the difference in the mean number of queries made by the application code to the underlying database. As can be noticed, the model does not explicitly account for connection management (e.g., establishing a new connection, recycling connections in a pool) overheads, and network latency. We assume that the overhead due to these issues to be negligible. Such an assumption is usually true when the number of calls made between the tiers to serve a single request is usually low (which is the case in TPC-App). However, if this is high (e.g., the maximum number of queries made by the code to generate a single page is as high as 80), then these overheads become sizeable and can reduce accuracy.

7.6.4. Validation with caches

As noted earlier, the model is explicitly designed to measure the end-to-end response time even when the application is deployed with caches at more than one tier. To evaluate the accuracy of the model when an application is deployed with caching tiers, we measured the performance of RUBBoS benchmark when a cache server is ran at the front-end tier or ahead of the database server. In these experiments, we present the result of these experiments.

RUBBoS with front-end cache

For our next step of experiments, as in the previous setup, we deployed the RUBBoS application on a single application server with a single back-end database.



(a) Mean Response Time **(b)** Standard Deviation

Figure 7.5: Response Times of RUBBoS Application with HTML Cache with a front-end cache server, an application server and a database server.

The configuration of this experiment corresponds to a system for which the mean response time is given by

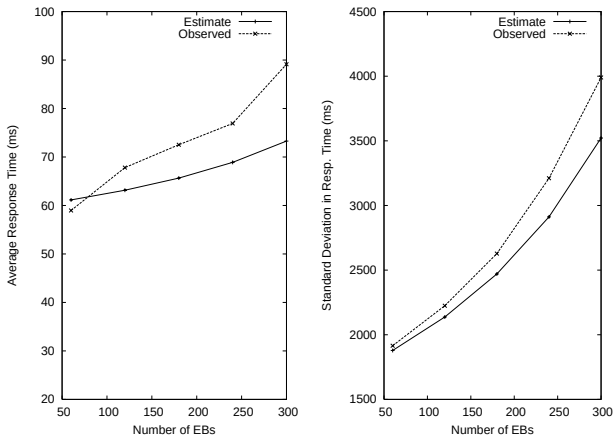
$$\mathbb{E}S = p_0\beta_{c,0} + (1-p_0)\left[\frac{(\mathbb{E}T_1+1)\beta_{s,0}}{1-\rho_{s,0}} + \mathbb{E}T_1\frac{\beta_{s,1}}{1-\rho_{s,1}}\right],$$

and the variance is approximated by

$$\mathbb{V}\text{ar}\, S \approx p_0^2\,\mathbb{V}\text{ar}\, S_{c,0} + (1-p_0)^2 \times \left[\mathbb{V}\text{ar}\, S_{s,0}(\mathbb{E}T_1+1) + \mathbb{V}\text{ar}\, S_{s,1}(1) + \mathbb{V}\text{ar}\, W_1\right].$$

In this experiment, we ran a front-end cache server that caches the HTML responses generated by the application code. Each client request is first received by the front-end cache which returns the response immediately if the corresponding response is available in the cache. Otherwise, the request is forwarded to the application server and the generated response is cached locally before returning it to the client. The consistency of the cached pages are maintained using a data dependency graph (e.g., [Challenger et al., 2005]) to determine the staleness of a cached page when the underlying database is updated.

In our experiments, we used the RUBBoS browse workload mix and varied the client workload by varying the number of EBs. All experiments were performed with a considerable warmup period at the start so that the hit ratio of the front-end caches was stable during the measurement period. We compared the mean



(a) Mean Response Time **(b)** Standard Deviation

Figure 7.6: Response Times of RUBBoS Application with an application server, a database cache server and a database server.

response times and its variance obtained from our experiments and the model. The results are given in Figure 7.5. As seen, the mean response times (and the standard deviation) of RUBBoS with HTML cache is significantly less compared to those of the simple RUBBoS system described in the previous section. This can be attributed due to the high cache hit rate (of 73.5%) at the front-end cache and the low execution times at the cache ($\beta_{c,0} = 2.76\,\text{ms}$). Moreover, similar to previous experiments, we can observe that the model predicts the response times of the application with a reasonably high accuracy with error margins usually less than 10%. We believe this is quite commendable considering that the characteristics of a cache server is completely different from that of the application or database server.

RUBBoS with database cache

In our final set of experiments, we evaluated the accuracy of the model for a system that employs a database cache. In this experiment, we deployed the RUBBoS application with a single application server, a database cache server (built using GlobeCBC) and a database server. The mean response time can be derived from our model as follows

$$\mathbb{E}S = \frac{(\mathbb{E}T_1 + 1)\beta_{s,0}}{1-\rho_{s,0}} + \mathbb{E}T_1\left[p_1\beta_{c,1} + (1-p_1)\frac{\beta_{s,1}}{1-\rho_{s,1}}\right],$$

and the variance is approximated by

$$\mathbb{V}\text{ar}\, S \approx \mathbb{V}\text{ar}\, S_{s,0}(\mathbb{E}T_1 + 1) + \left[p_1^2\, \mathbb{V}\text{ar}\, S_{c,1} + (1-p_1)^2 \mathbb{V}\text{ar}\, S_{s,1}(1) + \mathbb{V}\text{ar}\, W_1 \right].$$

All experiments were performed with a considerable warmup period at the start so that the hit ratio of the database cache was stable during the measurement period. The hit ratio of the database cache observed in our experiments was 82%. The mean execution time of the caching tier was 1.62 ms. The mean response times and its variance values obtained from our experiments and the model are shown in Figure 7.6. The model does a reasonable job once again. The error margins observed in our experiments were less than 12% for both mean and standard deviations.

7.6.5. Discussion

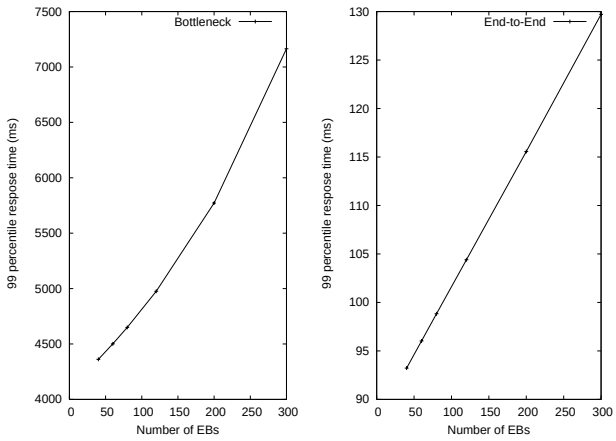
The above experiments demonstrate that the model does a commendable job in predicting the mean response times and its variances even for applications with vastly different characteristics deployed with different resource configurations. Our model is able to account for the effect of caching servers at different tiers accurately. Moreover, our experiments also suggest that a significant portion of the errors made by the model is due to ignoring the connection management overheads. As we showed in our experiments, connection management (such as waiting for connections in a connection pool) introduce additional delays and these tend to be significant when the number of interactions between the tiers is high (as in the case of RUBBoS). We plan to revise our model in the near future to accommodate these effects. To improve the accuracy of our predictions, we can use advanced profiling tools (e.g., [Barham et al., 2004]) which will allow us to measure the service time at each tier with higher degree of accuracy.

7.7. APPLICATIONS OF THE MODEL

In this section we discuss and demonstrate the benefits of the model to issues relating to resource provisioning, admission control, and service level agreement (SLA) negotiation.

7.7.1. Resource provisioning

As noted in previous chapter, for scalable hosting of multi-tiered Internet applications, the administrators need to determine the optimal number of resources to



(a) Bottleneck Analysis **(b)** End-to-End Analysis

Figure 7.7: 99 percentile response times of the RUBBoS Application obtained by (a) the bottleneck analysis method and (b) the end-to-end analysis method. Note that the scale of the y-axis in the figures differ by 3 orders of magnitude.

provision at each tier. As noted earlier, various techniques have been proposed that replicate the database, application or Web servers [Amiri et al., 2003a; Bornhövd et al., 2004; Chen et al., 2006; Doyle et al., 2003; Menasce, 2003; Plattner and Alonso, 2004; Rabinovich et al., 2003; Seltzsam et al., 2006]. However, to select the right set of techniques to apply we need to understand the impact of these systems and techniques on the end-to-end performance instead of the individual tiers. We believe our model can enable the administrators to determine the optimal resource configuration for a given application for different clients loads to meet a certain response time SLA.

The traditional way of resource provisioning (known as the bottleneck analysis method) is to identify the bottleneck tier and to improve its performance by caching or replicating it. Using our model, we propose to select the right resource configuration based on the end-to-end performance instead of individually optimizing each bottleneck tier. We call this the end-to-end analysis method.

Now, consider the scenario where a system administrator aims to provision the RUBBoS application (described in Section 7.6) such that the response time of 99% of the requests is below 200 ms. A commonly used technique is to equate the 99 percentile by $\mathbb{E}S + 3\sqrt{\mathbb{V}\text{ar}\,S}$ (popularly known as the 3-σ technique [Abraham, 2003]). For this scenario, using the bottleneck analysis method leads to the following conclusion. Based on the observation that 75% of the request-processing

time is spent in the back-end database server, the method concludes that the large portion of computation is performed by the database tier. To address this we can add a database cache or a database replica. However, from our previous experiments we can observe that the difference in service time of the database cache and the query execution time at the database server is only marginal. This is due to two reasons: (i) the overhead due to connection management, and JDBC resultset serialization and deserialization in the database cache node and (ii) the mean query execution time of RUBBoS application, as such, is very low. These results indicate that running a database cache for RUBBoS in not useful as database caching technologies are useful only if the database query execution time is significantly higher (as explained in [Sivasubramanian et al., 2006b; Olston et al., 2005]). Consequently, the bottleneck analysis method recommends adding a database replica (i.e., another server to the database tier) expecting it to reduce the load on the database tier by 50%. On the other hand, the end-to-end analysis recommends adding a front-end cache even though the application server is not the bottleneck. The performance of the resource configurations recommended by these two methods are given in Figure 7.7. As can be seen, the end-to-end analysis provides a superior performance compared to the bottleneck method yielding response times three order of magnitude less than the latter even at the 99 percentile while using the same number of resources. This can be explained by the ability of the model to capture the interrelationships between the different tiers.

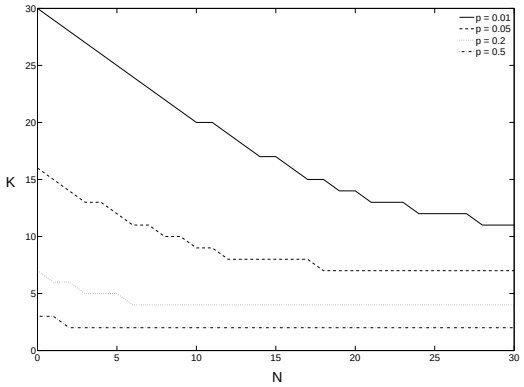
7.7.2. Admission control

The response times of an application may drop below its required target when many requests arrive at the entry nodes within a short period of time. To guarantee that the application continues to meet its performance targets, it needs to discriminate which requests are admitted into the network during these (transient) overload situations. In this section, we demonstrate how our model can be used to apply admission control to determine the point when and which requests to reject. To this end, consider the model of Section 7.3. We are interested in the mean response time when only K requests in total are admitted to the system, and requests in excess hereof are rejected.

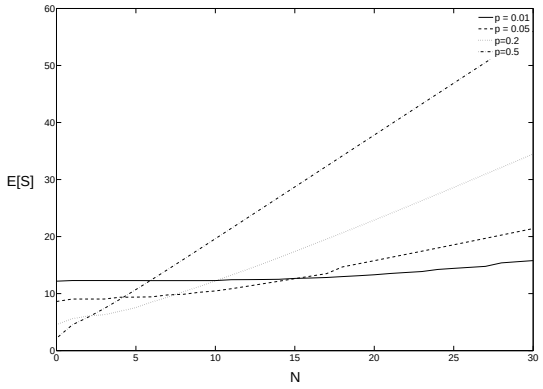
Let $\vec{l} = (l_{c,0}, \ldots, l_{c,N}, l_{s,0}, \ldots, l_{s,N})$ be the vector representing the number of requests at each tier in the application. As described in Section 7.4, the stationary probability of having configuration $\vec{l}$, when no admission control is applied, is given by $q(\vec{l}) = \prod_{i=0}^{N} \mathbb{P}(L_{c,i} = l_{c,i}) \prod_{i=0}^{N} \mathbb{P}(L_{s,i} = l_{s,i})$. Define q_i by

$$q_i = \sum_{\{\vec{l} \mid l_{c,0}+\cdots+l_{c,N}+l_{s,0}+\cdots+l_{s,N}=i\}} q(\vec{l}).$$

Then, the probability of having configuration $\vec{l}$, when admission control is applied,



(a) Iso-loss curves



(b) Delay curves

Figure 7.8: Iso-loss curves for several blocking probabilities p with associated delay curves.

is represented by $q(\vec{l})/\sum_{i=1}^{K} q_i$. In particular, the probability of having in total l customers in the system can be written as

$$\mathbb{P}(L = l) = \frac{q_l}{\sum_{i=1}^{K} q_i}.$$

Using this expression and the arrival theorem by Lavenberg and Reiser [Lavenberg and Reiser, 1980], one can obtain the mean response time $\mathbb{E}S_{\mathrm{ac},0}$, when applying

admission control with no buffers[1]. This yields

$$\mathbb{E}S_{\text{ac},0} = \frac{\sum_{l=1}^{K} \frac{l}{\lambda} q_l}{\sum_{i=0}^{K-1} q_i}.$$

A similar approximate analysis can be performed when a buffer of size N is added in front of the application. When K customers are already in the system, the requests in excess of K are buffered and wait their turn to enter the system in an FCFS manner. However, when the buffer is full, requests are again rejected. The mean response time $\mathbb{E}S_{\text{ac},N}$, when admission control is applied with a buffer size of N, is then approximated by

$$\mathbb{E}S_{\text{ac},N} \approx \mathbb{P}(L^{(0)} = 0) q_K \sum_{l=0}^{N} l \Big(\frac{q_K}{q_{K-1}}\Big)^{l+1} + \mathbb{P}(L^{(1)} = 0) \Bigg[\sum_{l=1}^{K} \frac{l}{\lambda} q_l + \frac{K}{\lambda} q_K \sum_{j=1}^{N} \Big(\frac{q_K}{q_{K-1}}\Big)^{j} \Bigg],$$

where the expression $\mathbb{P}(L^{(a)} = 0)$ is defined as

$$\mathbb{P}(L^{(a)} = 0) = \Bigg[\sum_{l=0}^{K} q_l + q_K \sum_{j=1}^{N-a} \Big(\frac{q_K}{q_{K-1}}\Big)^{j} \Bigg]^{-1}.$$

We can study the influence of the buffer size using the results for admission control systems with and without a buffer given in the previous paragraphs. Figure 7.8(a) shows the influence of the buffer size N as iso-loss curves, i.e., all combinations of N and K for which the blocking probability $\mathbb{P}(L^{(0)} = N + K)$ is a constant p. The results are for a 3-tier system with one service tier at the front-end, and two multi-server FCFS service tiers at the back-end. The parameters used for this model are $\lambda = 1$, $\beta_{s,0} = 0.3$, $\beta_{s,1} = 1.6$, and $\beta_{s,2} = 0.9$.

The results show that only three customers are allowed in the system to obtain a blocking probability of 0.5 when the buffer size is less than two. When the buffer size is larger than two, only two customers are allowed. Consequently, to guarantee that the blocking probability is less than 0.5, the threshold K must be greater than three when the buffer size N is less than two, and otherwise, K must be greater than two. Since differences in the buffer size and threshold limit influence the mean response time, Figure 7.8(b) shows the delay curves corresponding to the values of the iso-loss curves as a function of N. Thus, for instance, Figure 7.8(b) shows that for a blocking probability of 0.5, the mean response time increases linearly when the buffer size N increases.

[1]Here, buffers describe waiting queues that hold unprocessed requests yet to be admitted to the system.

The graphs of Figure 7.8 show the importance of selecting an appropriate buffer size with its corresponding blocking probability. When the blocking probability is high, the number of allowed customers in the application will be small, so that queueing mainly occurs in the buffer in front of the application. When the blocking probability is small, the buffer in front of the application is hardly used, so the number of admitted requests is high with as result that queueing occurs mainly within the application.

7.7.3. SLA negotiation

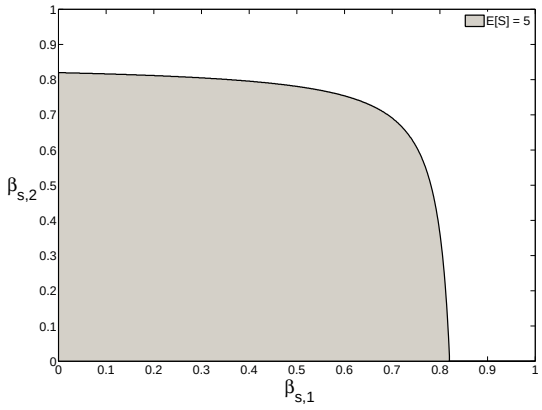
In this section, we demonstrate how to compute performance measures that can be used in effective Service Level Agreement (SLA) negotiation. SLA negotiation is a common problem that arises when an application is composed of not just its own business logic and databases but also using services provided by external companies. For example, many small Internet retailers use amazon.com's Web service storage service[2], order fulfillment service[3] to build their own application.

Now, let us see how our model can be used to perform these SLA negotiations. Consider a 3-tier system with one service tier at the front-end, called the application server (AS), and two external service tiers that are part of different domains. Assume that the AS has an SLA with its end-users stating that the mean duration for obtaining a response is less than 5 seconds with a maximum standard deviation of 2 seconds. Moreover, suppose that requests of users arrive according to a Poisson process at the AS with rate $\lambda = 1$. The AS has a mean service time of $\beta_{s,0} = 0.1$ seconds, and the external service tiers are single server FCFS nodes with exponentially distributed service times. The key question for effective SLA negotiation is: "*What combination of SLAs with the other domains leads to the desired response times?*". More specifically, one could ask: What is the SLA negotiation space of the AS?

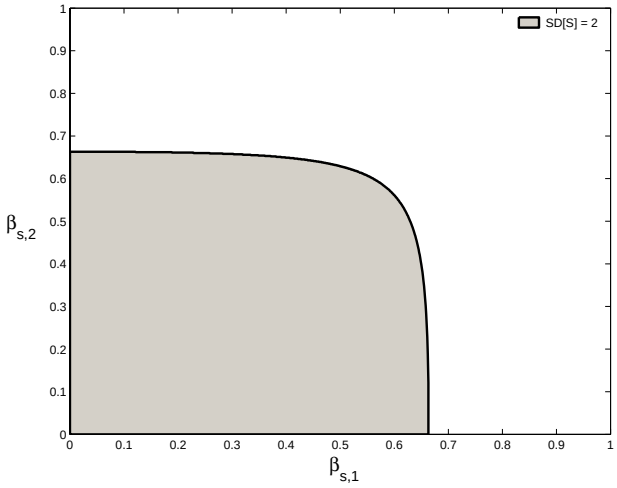
Mean response times: The total response time can be split up in the sojourn time $S_{s,0}$ at the AS, and the sojourn times $S_{s,1}$ and $S_{s,2}$ at the external service tiers. The results of Section 7.4 yield that the mean sojourn time at the AS equals $\mathbb{E}S_{s,0} = 0.43$ seconds. Since the mean response time is constrained by 5 seconds, a request can spend on average up to a maximum of 4.57 seconds at the other services. The other services handle just one request at a time, i.e., if there are more requests at those service tiers, requests have to wait and will be served according to an FCFS discipline. The results in Section 7.4 enable us to compute all combinations of the mean service times $\beta_{s,1}$ and $\beta_{s,2}$ for which the total response time is at most 4.57 seconds.

[2] http://www.amazon.com/s3

[3] http://www.amazonservices.com/fulfillment/



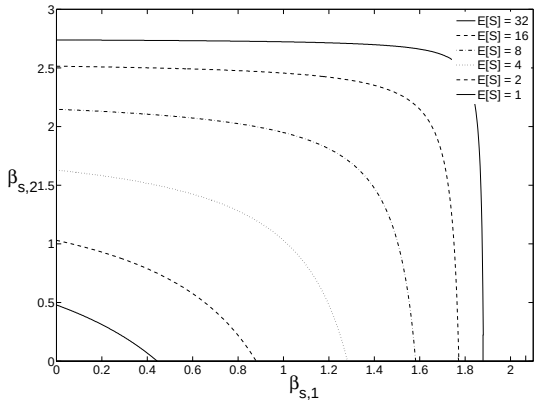
(a) Negotiation space under the constraint $\mathbb{E}[S] < 5$



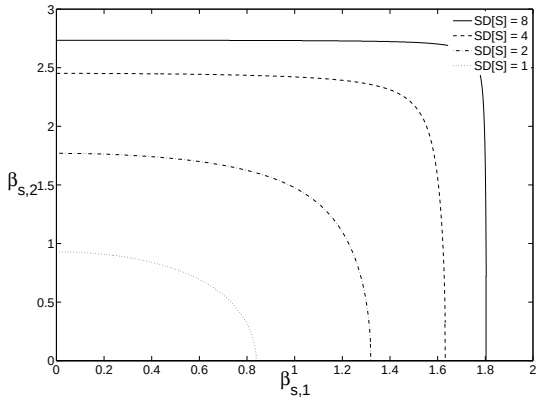
(b) Negotiation space under the constraint $\mathrm{SD}[S] < 2$

Figure 7.9: The SLA negotiation space for all $\beta_{s,1}$ and $\beta_{s,2}$.

On first glance, one would expect that the region of these service times is bounded by a linear curve, since higher service times at one tier must result in lower service times at the other. Figure 7.9(a) shows that this is not the case, and that the boundary is non-linear. This negotiation space, together with the costs of providing a specific service time, can be utilized by the AS to minimize the total costs by negotiating an SLA with both service tiers using the optimal $\beta_{s,1}$ and $\beta_{s,2}$.



(a) Iso-curves for the mean $\mathbb{E}[S]$



(b) Iso-curves for the standard deviation $\mathrm{SD}[S]$

Figure 7.10: The SLA negotiation space for all $\beta_{s,1}$ and $\beta_{s,2}$.

Standard deviation of the response times: By applying the same procedure as in the previous paragraphs, we can determine the negotiation space of the AS with the restriction that the standard deviation of the total sojourn time is at most 2 seconds. By using the results of Section 7.4, we obtain the variance $\mathbb{V}\mathrm{ar}\, S_{s,0}$ of the sojourn time at the AS. Subsequently, we can compute all combinations of $\beta_{s,1}$ and $\beta_{s,2}$ such that their standard deviation, when added to $\mathbb{V}\mathrm{ar}\, S_{s,0}$, is at most 2 seconds. Figure 7.9(b) shows the region of these service times. To adhere to

the two restrictions simultaneously, i.e., the expectation of the sojourn time is at most 5 seconds and the standard deviation is at most 2 seconds, we intersect both regions to keep the smallest region with combinations of $\beta_{s,1}$ and $\beta_{s,2}$.

In order to study the effects of the constraints on the parameter regions, we have constructed iso-curves for the upper bounds on the mean $\mathbb{E}S$ and the standard deviation SD[S]. Figure 7.10(a) and (b) show these iso-curves that depict all combinations of $\beta_{s,1}$ and $\beta_{s,2}$ such that $\mathbb{E}S$ and SD[S], respectively, have a specific constant value. The figure shows that a more strict constraint on $\mathbb{E}S$ results in a more linear iso-curve. In such a case, the service times are that low so that the sojourn times almost equal the service times, and therefore almost no queueing occurs. The case for a less strict constraint results in an iso-curve that almost has a square shape. This is explained by the fact that the longer requests have to wait in one queue of the network, the shorter they have to wait in the other queue as long as the service time of the second queue is less than the service time of the first queue.

7.8. CONCLUSION

In this chapter, we presented an analytical model for multi-tiered Internet applications. Based on this model, we can estimate the mean response time of an application and also provide accurate approximations to its variance. We verified the effectiveness of our approximations using numerical simulations resulting in a margin of error of less 6%. Subsequently, we performance extensive experimentations with multiple industry-standard Web application benchmarks for a wide-range of resource configurations. Our experiments we observed that the error margin of the model is usually less than 10%. This demonstrates the accuracy of our model to capture the real behaviour of for a wide range of Internet applications.

To the best of our knowledge, this is the first work that aims to model the variability in response times for multi-tiered Internet applications. Our proposed model is highly flexible as it allows to easily accommodate for changes in the resource configuration. Moreover, the model can also deal with applications deployed multiple caching tiers with high accuracy. We also demonstrated how our model can be applied to SLA-driven resource provisioning, effective admission control, and SLA negotiation. We believe that our model is highly beneficial since it deals with many realistic design problems that are faced by the designers and administrators of modern Internet systems.

CHAPTER 8

Conclusion

Modern Web sites have evolved from simple monolithic systems to complex multi-tiered systems. In contrast to traditional Web sites, these sites do not simply deliver pre-written content but dynamically generate content using (one or more) multi-tiered Web applications. In this thesis, we addressed the question: *How to host multi-tiered Web applications in a scalable manner?*

Scaling up a Web application requires scaling its individual tiers. To this end, various research works have proposed techniques that employ replication or caching solutions at different tiers. However, most of these techniques aim to optimize the performance of individual tiers and not the entire application. A key observation made in our research is that there exists no elixir technique that performs the best for all Web applications. Effective hosting of a Web application requires careful selection and deployment of several techniques at different tiers. In essence, for scalable hosting of multi-tiered applications one must:

"Think global, act local."

From the results presented in this thesis, we can infer that scalable hosting of Web applications requires: (i) *local actions:* scalability techniques to be employed at each tier (of an application) to improve its individual performance, and (ii) *global perspective:* an end-to-end evaluation approach that allows us to select the right set of local actions (i.e., scalability techniques) to employ for a given application such that its end-to-end performance is maximized with minimum hosting costs.

Local actions

As noted above, local actions at each tier are essential to improve the performance of the individual tiers. For instance, a simple Web application usually consists

of three tiers: presentation, business logic, and database. Techniques to improve the scalability of the presentation tier of these applications are well understood and successfully employed by CDNs to host static Web content (as described in Chapter 2). However, scaling up the business-logic tier involves replication of the application code across multiple servers. This problem is also well understood provided that the application code is stateless.

Scaling up the database tier is harder and has a significant impact on the performance of database-driven Web applications. As we demonstrated in this thesis, for data-intensive applications, mere replication of the application code at the edge servers and a centralized database system often does not suffice, as generation of each page still requires the application code to make many queries to the database. To this end, we propose a database caching technique, GlobeCBC, that improves the performance of Web applications provided the application's query workload exhibits high locality (see Chapter 3). While GlobeCBC is suited mostly for applications with high query locality, applications that do not have these characteristics require replication of the application data to have low access latencies to the database. To this end, we presented GlobeDB, a system that performs autonomic replication of application data (See Chapter 4). In Chapter 5, we demonstrated how the database throughput (in addition to its response time) can be improved by a combination of partial replication, query caching and query routing.

Global perspective

While the aforementioned techniques and systems improve the performance of the individual tiers (and eventually the application), an application's administrator is not only interested in the performance of its individual tiers but also in its end-to-end performance. However, due to the complex interaction between the tiers, tier-level optimization does not allow us to explain the overall performance of the application. This calls for a model that allows us to understand the *global end-to-end performance* of an application in relation to the performance of the individual tiers and provision resources accordingly (to meet its performance-related SLAs).

To this end, we propose a resource provisioning approach (described in Chapter 6) that allows us to choose the best resource configuration for hosting a Web application such that its end-to-end response time can be optimized with minimum usage of resources. The proposed approach is based on an analytical model for multi-tier systems (described in Chapters 6 and 7), which allows us to derive expressions for estimating the mean end-to-end response time and its variance. The proposed model is also capable of modeling the caching techniques that can be incorporated at different tiers. We validate the accuracy of the model and the effectiveness of our resource provisioning approach through extensive experimentation with several industry-standard benchmarks. Our evaluations suggest that our

proposed resource provisioning approach makes the correct choices for different types of services and workloads. In many cases, our approach reaches the required performance-related SLA with less servers than traditional resource provisioning techniques.

Future directions

This thesis presented several techniques and approaches to improve the scalability of multi-tiered Web applications. Undoubtedly, this is a large area that cannot be fully covered in a single dissertation. There are a number of directions in which our research can be extended or complemented.

There are a few open research issues that arise from the analytical model we employed in Chapters 6 and 7. First, the results presented in this thesis were validated with only industry-standard benchmarks and not with real-world traces. We believe performing validations with real-world traces can give us more insights and will allow us to refine the model, if necessary. Second, the proposed model allows us to study the system under steady state and cannot estimate the end-to-end response time during overloads. Modeling end-to-end response times during overloads (i.e., when the arrival rate of requests is greater than the departure rate) requires different analytical models altogether. We believe such a model is essential for online businesses as they are interested in their application's performance not only during stable loads but also during overloads. Finally, deriving expressions for estimating the percentiles (e.g., 99%, 99.9%) in end-to-end response times would be desirable.

In the context of the SLA-driven resource provisioning method (described in Chapter 6), it would be interesting to investigate adaptation mechanisms that must be employed in the face of flash crowds (sudden and huge increase in request rate). For handling such events, we envisage that the adaptations to be performed might require mechanisms of different nature such as enhanced admission control, and reduced consistency requirements.

Most of the results presented in this thesis assume a failure-free environment. Server failures in a tier can be tolerated easily as long as the tier is stateless [Brewer, 2000]. However, handling failures in the database tier is harder. As proved in [Gilbert and Lynch, 2002], it is impossible to provide both strong consistency and perfect availability in a system that is prone to server failures and network partitions.

If we restrict ourselves to server failures, we believe that traditional database replication solutions (used in cluster environments) can be employed to improve the availability of the database tier. However, when GlobeCBC-based database query caching is employed, we must address two new availability related issues. First, when an edge server fails and its clients are redirected to another edge

server, we must make sure that the failover remains transparent to the clients. This requires to provide read-your-write consistency guarantees. Second, when the origin database server fails, another one must be ready to take over. While maintaining the availability of the origin database can be reduced to a classical replicated database problem, we must also make sure that no cache invalidation is lost so that application consistency is preserved.

We have addressed these issues in [Rilling et al., 2007] where we show that these two questions can be answered with a few changes to GlobeCBC and that the resulting performance overhead is low. However, it would be interesting to extend these mechanisms such that they can tolerate network partitions and examine its impact on consistency.

Security related issues have not been addressed in this thesis. The replication and caching techniques presented in this thesis are designed for a trusted environment where all servers belong to a single organization. However, if the servers are formed out of resources from multiple organizations as done in collaborative CDNs like Globule [Pierre and van Steen, 2006], security and privacy of the replicated data become important issues and are very much open problems.

To host Web applications in an untrusted environment, we need to address three important issues. First, we need to design security-aware data placement algorithms that allow application providers to define and select "trustable" servers to host their applications (e.g., [Crispo et al., 2005]). Second, we need mechanisms to enable the application provider to verify the integrity of the responses generated by the third party servers (e.g., [Popescu et al., 2005]). Third, we need privacy-aware data replication mechanisms to ensure that third-party servers do not access application data they are not authorized to access (e.g., [Manjhi et al., 2007]).

Summary

The ever-growing popularity of the Web has forced many businesses to open their processes to their Web clients. The software systems hosting these businesses are exceedingly complex. For instance, a single client response is generated by (tens or hundreds of) multi-tiered applications [Vogels, 2006]. The client-perceived performance of these applications is crucial for the sustainability of these businesses. In this thesis, we have presented several techniques and systems that not only can improve the performance of individual tiers but also can aid the administrators of these systems in selecting the best set of techniques to host their applications. We believe these techniques can aid and help Web practitioners in taming the problem of scalable hosting of multi-tiered Web applications.

SAMENVATTING

Schaalbare Exploitatie van Web Applicaties

Moderne Web sites zijn geëvolueerd van simpele monolithische systemen naar complexe meerlagen systemen. In tegenstelling tot traditionele Web sites leveren deze sites niet simpelweg bestaande pagina's maar genereren ze pagina's dynamisch met behulp van (één of meerdere) meerlagige Web applicaties. Deze dissertatie gaat in op de volgende onderzoeksvraag: *Hoe kan een meerlagige Web applicatie op een schaalbare en efficiënte manier geëxploiteerd worden?*

Het opschalen van een Web applicatie vereist het opschalen van zijn individuele lagen. Hiertoe hebben verscheidene onderzoeken technieken voorgesteld die replicatie of *caching* oplossingen toepassen in de verschillende lagen. Een sleutelobservatie uit ons onderzoek is dat er geen enkele techniek bestaat die het beste werkt voor alle Web applicaties. Effectieve exploitatie van een Web applicatie vereist zorgvuldige selectie en toepassing van meerdere technieken in verschillende lagen.

De onderliggende these van deze dissertatie is dat schaalbare exploitatie van Web applicaties vereist: (1) *lokale acties:* schaalbaarheidstechnieken moeten toegepast worden in iedere laag om zijn individuele prestaties te verbeteren, en (2) een *globaal perspectief:* een evaluatie aanpak op het hoogste systeemniveau die ons in staat stelt de juiste verzameling van lokale acties (d.w.z. schaalbaarheidstechnieken) te kiezen voor een gegeven applicatie zodat eindprestaties worden gemaximaliseerd met minimale exploitatie kosten.

Lokale Acties

Een simpele Web applicatie bestaat normaliter uit drie lagen: presentatie, business logica en databank. Technieken om de schaalbaarheid van de presentatielaag te

verbeteren zijn uitonderzocht en worden succesvol toegepast in zogeheten *content-delivery networks* om statische Web pagina's aan te bieden (zoals beschreven in Hoofdstuk 2). Het opschalen van de business logica-laag daarentegen vereist het repliceren van de applicatiecode over meerdere servers. Dit probleem is ook uitonderzocht in het geval dat de applicatiecode geheugenloos is.

Het opschalen van de databanklaag is moeilijker en heeft een significante uitwerking op de prestaties van databankgedreven Web applicaties. Voor data-intensieve applicaties is het slechts repliceren van de applicatiecode op de *edge* servers en een centraal databanksysteem onvoldoende. Dit omdat voor het genereren van elke pagina de applicatie code nog steeds meerdere zoekopdrachten moet uitvoeren op de databank. Hiertoe introduceren we een *caching* techniek voor databanken voor, genaamd GlobeCBC, die de prestaties van Web applicaties verbeterd onder de voorwaarde dat de zoekopdrachten van de applicatie een hoge lokaliteit vertonen. Applicaties die niet aan deze voorwaarde voldoen vereisen replicatie van de applicatiedata om lage toegangstijden tot de databank te realiseren. Voor deze applicaties presenteren we GlobeDB, een systeem voor autonome replicatie van applicatiedata (zie Hoofdstuk 4). In Hoofdstuk 5 laten we zien hoe, naast de reactietijd, ook de doorvoersnelheid van de databank verbeterd kan worden door een combinatie van partiële replicatie, *caching* van zoekopdrachten en het routeren van zoekopdrachten.

Globaal Perspectief

Hoewel de zojuist genoemde technieken en systemen de prestaties van de individuele lagen (en uiteindelijk dus die van de applicatie) verbeteren is een applicatiebeheerder niet slechts geïntereseerd in de prestaties van de individuele lagen, maar ook in de prestaties van de applicatie als geheel gemeten. De complexe interactie tussen lagen stelt ons echter niet in staat de uiteindelijke prestaties van de applicatie te verklaren met de optimalisaties per laag. Dit vraagt om een model dat ons in staat stelt de *globale prestaties* van een applicatie te begrijpen in relatie tot de prestaties van de individuele lagen, en dienovereenkomstig middelen te alloceren zodat deze aan zijn prestatie-gerelateerde *Service-Level Agreements (SLAs)* kan voldoen.

We introduceren hiertoe een aanpak voor de allocatie van middelen voor (beschreven in Hoofdstuk 6) die ons de beste configuratie van middelen laat kiezen voor de exploitatie van een Web applicatie, zodat zijn uiteindelijke reactietijd geoptimaliseerd kan worden met een minimaal gebruik van middelen. De voorgestelde aanpak is gebaseerd op een analytisch model voor meerlagige systeem (beschreven in Hoofdstuk 6 en 7), dat ons in staat stelt formules af te leiden voor het schatten van de gemiddelde reactietijd en de variantie daarvan. Het voorgestelde model is ook in staat om de *caching* technieken die in de verschillende

lagen toegepast kunnen worden te modelleren. We valideren de accuratesse van het model en de effectiviteit van onze aanpak voor de allocatie van middelen door uitgebreide experimenten met standaard maatstaven uit de industrie. Onze evaluaties suggereren dat onze voorgestelde aanpak de juiste keuzes maakt voor verschillende diensten en systeembelastingen. In veel gevallen haalt onze aanpak de vereiste prestatie-gerelateerde SLA met minder servers dan traditionele allocatie technieken.

BIBLIOGRAPHY

Abdelzaher, T. F., Shin, K. G., and Bhatti, N. (2002). Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96.

Aboba, B., Arkko, J., and Harrington, D. (2000). Introduction to Accounting Management. RFC 2975.

Abraham, B. (2003). Quality improvement, six sigma, and statistical thinking. *Stat. Methods*, 5(2):41–56.

Aggarwal, A. and Rabinovich, M. (1998). Performance of Replication Schemes for an Internet Hosting Service. Technical Report HA6177000-981030- 01-TM, AT&T Research Labs, Florham Park, NJ.

Amiri, K., Park, S., Tewari, R., and Padmanabhan, S. (2003a). DBProxy: A dynamic data cache for web applications. In *Proceedings of Internation Conference on Data Engineering*, pages 821–831.

Amiri, K., Park, S., Tewari, R., and Padmanabhan, S. (2003b). Scalable template-based query containment checking for web semantic caches. In *Proceedings of Internation Conference on Data Engineering*, pages 493–504.

Amiri, K., Tewari, R., Park, S., and Padmanabhan, S. (2002). On space management in a dynamic edge data cache. In *WebDB*, pages 37–42.

Amza, C., Cox, A., and Zwaenepoel, W. (2003). Conflict-aware scheduling for dynamic content applications. Proceedings of the Fifth USENIX Symposium on Internet Technologies and Systems.

Amza, C., Soundararajan, G., and Cecchet, E. (2005). Transparent caching with strong consistency in dynamic content web sites. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 264–273, New York, NY, USA. ACM Press.

Andrews, M., Shepherd, B., Srinivasan, A., Winkler, P., and Zane, F. (2002). Clustering and Server Selection Using Passive Monitoring. In *21st INFOCOM Conference*, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Ardaiz, O., Freitag, F., and Navarro, L. (2001). Improving the Service Time of Web Clients using Server Redirection. In *2nd Workshop on Performance and Architecture of Web Servers*, New York, NY. ACM, ACM Press.

Arlitt, M., Krishnamurthy, D., and Rolia, J. (2001). Characterizing the scalability of a large web-based shopping system. *ACM Transactions on Internet Technology*, 1(1):44–69.

Ballintijn, G., van Steen, M., and Tanenbaum, A. (2000). Characterizing Internet Performance to Support Wide-area Application Development. *Operating Systems Review*, 34(4):41–47.

Barbir, A., Cain, B., Douglis, F., Green, M., Hoffman, M., Nair, R., Potter, D., and Spatscheck, O. (2002). Known CDN Request-Routing Mechanisms. Work in progress.

Barford, P., Cai, J.-Y., and Gast, J. (2001). Cache Placement Methods Based on Client Demand Clustering. Technical Report TR1437, University of Wisconsin at Madison.

Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. (2004). Using magpie for request extraction and workload modelling. In *Proceedings of USENIX Operating Systems Design and Implementation*.

Barroso, L., Dean, J., and Hlzle, U. (2003). Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28.

Baskett, F., Chandy, K. M., Muntz, R. R., and Palacios, F. G. (1975). Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260.

Bennani, M. N. and Menasce, D. A. (2005). Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 229–240, Washington, DC, USA. IEEE Computer Society.

Berg, J. v. d. and Boxma, O. (1991). The M/G/1 queue with processor sharing and its relation to a feedback queue. *Queueing Syst. Theory Appl.*, 9(4):365–402.

Bernstein, P. A. and Goodman, N. (1983). The failure and recovery problem for replicated databases. In *2nd Symposium on Principles of Distributed Computing*, pages 114–122, New York, NY. ACM Press.

Bhide, M., Deolasee, P., Katkar, A., Panchbudhe, A., Ramamritham, K., and Shenoy, P. (2002). Adaptive Push-Pull: Disseminating Dynamic Web Data. *IEEE Transactions on Computers*, 51(6):652–668.

Bloom, B. H. (1970). Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426.

Bornhövd, C., Altinel, M., Mohan, C., Pirahesh, H., and Reinwald, B. (2004). Adaptive database caching with DBCache. *Data Engineering*, 27(2):11–18.

Boxma, O. and Daduna, H. (1990). Sojourn times in queueing networks. *Stochastic Analysis of Computer and Communication Systems*, pages 401–450.

Boxma, O., Mei, R. v. d., Resing, J., and Wingerden, K. v. (2005). Sojourn time approximations in a two-node queueing network. In *Proceedings of the 19th International Teletraffic Congress - ITC 19*, pages 1121–1133.

Brewer, E. A. (2000). Towards robust distributed systems (abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, New York, NY, USA. ACM Press.

Brynjolfsson, E., Smith, M., and Hu, Y. (2003). Consumer surplus in the digital economy: Estimating the value of increased product variety at online booksellers. MIT Sloan Working Paper No. 4305-03.

Cao, P. and Irani, S. (1997). Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA.

Cao, P. and Liu, C. (1998). Maintaining Strong Cache Consistency in the World Wide Web. *IEEE Transactions on Computers*, 47(4):445–457.

Cao, P., Zhang, J., and Beach, K. (1998). Active cache: Caching dynamic contents on the Web. In *Proceedings of the Middleware Conference*, pages 373–388.

Cardellini, V., Colajanni, M., and Yu, P. (1999). Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing*, 3(3):28–39.

Carter, R. L. and Crovella, M. E. (1997). Dynamic Server Selection Using Bandwidth Probing in Wide-Area Networks. In *16th INFOCOM Conference*, pages 1014–1021, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Castro, M., Costa, M., Key, P., and Rowstron, A. (2003). PIC: Practical Internet Coordinates for Distance Estimation. Technical Report MSR-TR-2003-53, Microsoft Research.

Cate, V. (1992). Alex – A Global File System. In *File Systems Workshop*, pages 1–11, Berkeley, CA. USENIX, USENIX.

Cecchet, E. (2004). C-JDBC: a middleware framework for database clustering. *Data Engineering*, 27(2):19–26.

Challenger, J., Dantzig, P., Iyengar, A., and Witting, K. (2005). A fragment-based approach for efficiently creating dynamic web content. *ACM Transactions on Internet Technology*, 5(2):359–389.

Chandra, P., Chu, Y.-H., Fisher, A., Gao, J., Kosak, C., Ng, T. E., Steenkiste, P., Takahashi, E., and Zhang, H. (2001). Darwin: Customizable Resource Management for Value-Added Network Services. *IEEE Network*, 1(15):22–35.

Chen, J., Soundararajan, G., and Amza, C. (2006). Autonomic provisioning of backend databases in dynamic content web servers. In *Proc. Intl Conf. on Autonomic Computing*.

Chen, Y., Katz, R., and Kubiatowicz, J. (2002a). Dynamic Replica Placement for Scalable Content Delivery. In *1st International Workshop on Peer-to-Peer Systems*.

Chen, Y., Qiu, L., Chen, W., Nguyen, L., and Katz, R. H. (2002b). Clustering web content for efficient replication. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP'02)*, pages 165–174.

Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. (2005). Live migration of virtual machines. In *Proc. NSDI Symposium*.

Coffman, E., Muntz, R., and Trotter, H. (1970). Waiting time distributions for processor-sharing systems. *Journal of the ACM*, 17(1):123–130.

Cohen, E. and Kaplan, H. (2001). Proactive Caching of DNS Records: Addressing a Performance Bottleneck. In *1st Symposium on Applications and the Internet*, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Conti, M., Gregori, E., and Lapenna, W. (2002). Replicated Web Services: A Comparative Analysis of Client-Based Content Delivery Policies. In *Networking 2002 Workshops*, volume 2376 of *Lecture Notes on Computer Science*, pages 53–68, Berlin. Springer-Verlag.

Cooper, R. (1981). *Introduction to Queueing Theory*. North Holland.

Cox, R., Dabek, F., Kaashoek, F., Li, J., and Morris, R. (2003). Practical, Distributed Network Coordinates. In *2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, MA, USA.

Crispo, B., Sivasubramanian, S., Mazzoleni, P., and Bertino, E. (2005). P-hera: Scalable fine-grained access control for p2p infrastructures. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pages 585–591, Washington, DC, USA. IEEE Computer Society.

Crovella, M. and Carter, R. (1995). Dynamic Server Selection in the Internet. In *3rd Workshop on High Performance Subsystems*, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

da Cunha, C. R. (1997). Trace Analysis and its Applications to Performance Enhancements of Distributed Information Systems. Ph.D. Thesis, Boston University.

Dar, S., Franklin, M. J., Jonsson, B., Srivastava, D., and Tan, M. (1996). Semantic data caching and replacement. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341.

Datta, A., Dutta, K., Thomas, H., VanderMeer, D., Suresha, and Ramamritham, K. (2002). Proxy-based acceleration of dynamically generated content on the world wide web: an approach and implementation. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 97–108. ACM Press.

Davis, A., Parikh, J., and Weihl, W. E. (2004). Edgecomputing: extending enterprise applications to the edge of the internet. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 180–187, New York, NY, USA. ACM Press.

Delgadillo, K. (1999). Cisco DistributedDirector. Technical report, Cisco Systems, Inc.

Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B. (2002). Globally Distributed Content Delivery. *IEEE Internet Computing*, 6(5):50–58.

Douglis, F., Haro, A., and Rabinovich, M. (1997). HPP: HTML macro-preprocessing to support dynamic document caching. In *USENIX Symposium on Internet Technologies and Systems*.

Doyle, R., Chase, J., Asad, O., Jin, W., and Vahdat, A. (2003). Web server software architectures. In *Proc. USENIX Symposium on Internet Technologies and Systems*.

Duvvuri, V., Shenoy, P., and Tewari, R. (2000). Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *19th INFOCOM Conference*, pages 834–843, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Dykes, S. G., Robbins, K. A., and Jeffrey, C. L. (2000). An Empirical Evaluation of Client-side Server Selection. In *19th INFOCOM Conference*, pages 1361–1370, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Elnikety, S., Zwaenepoel, W., and Pedone, F. (2005). Database replication using generalized snapshot isolation. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, Washington, DC, USA. IEEE Computer Society.

Fei, Z. (2001). A Novel Approach to Managing Consistency in Content Distribution Networks. In *6th Web Caching Workshop*, Amsterdam. North-Holland.

Fei, Z., Bhattacharjee, S., Zegura, E. W., and Ammar, M. H. (1998). A novel server selection technique for improving the response time of a replicated service. In *Proceedings of INFOCOM*, pages 783–791.

Fitzpatrick, B. (2004). Inside LiveJournal's backend, or "holy hell that's a lot of hits!". Presentation at the O'Reilly Open Source Convention. http://www.danga.com/words/2004_oscon/oscon2004.pdf.

Francis, P., Jamin, S., Jin, C., Jin, Y., Raz, D., Shavitt, Y., and Zhang, L. (2001). IDMaps: Global Internet Host Distance Estimation Service. *IEEE/ACM Transactions on Networking*, 9(5):525–540.

Francis, P., Jamin, S., Paxson, V., Zhang, L., Gryniewicz, D., and Jin, Y. (1999). An Architecture for a Global Internet Host Distance Estimation Service. In *18th INFOCOM Conference*, pages 210–217, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Fu, Y., Cherkasova, L., Tang, W., and Vahdat, A. (2002). EtE: Passive End-to-End Internet Service Performance Monitoring. In *USENIX Annual Technical Conference*, pages 115–130, Berkeley, CA. USENIX, USENIX.

Gao, L., Dahlin, M., Nayate, A., Zheng, J., and Iyengar, A. (2003). Application Specific Data Replication for Edge Services . In *12th International World Wide Web Conference*, New York, NY. ACM, ACM Press.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *Proc. SOSP*, pages 29–43.

Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59.

Gray, C. and Cheriton, D. (1989). Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *12th Symposium on Operating System Principles*, pages 202–210, New York, NY. ACM, ACM Press.

Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, San Mateo, CA.

Gummadi, K. P., Saroiu, S., and Gribble, S. D. (2002). King: Estimating Latency between Arbitrary Internet End Hosts. In *2nd Internet Measurement Workshop*, pages 5–18, New York, NY. ACM, ACM Press.

Holliday, J., Agrawal, D., and Abbadi, A. (2002). Partial database replication using epidemic communication. In 22nd International Confernce on Distributed Computing Systems (ICDCS), pages 485–493, Vienna, Austria, July 2002.

Huffaker, B., Fomenkov, M., Plummer, D. J., Moore, D., and Claffy, K. (2002). Distance Metrics in the Internet. In *International Telecommunications Symposium*, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Hull, S. (2002). *Content Delivery Networks*. McGraw-Hill, New York, NY.

Jackson, J. (1957). Networks of waiting lines. *Operations Research*, 5:518–521.

Janiga, M. J., Dibner, G., and Governali, F. J. (2001). Internet Infrastructure: Content Delivery. Goldman Sachs Global Equity Research.

Johnson, K. L., Carr, J. F., Day, M. S., and Kaashoek, M. F. (2001). The Measured Performance of Content Distribution Networks. *Computer Communications*, 24(2):202–206.

Jung, J., Krishnamurthy, B., and Rabinovich, M. (2002a). Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *11th International World Wide Web Conference*, pages 252–262.

Jung, J., Krishnamurthy, B., and Rabinovich, M. (2002b). Flash crowds and denial of service attacks: characterization and implications for cdns and web sites. In *Proc. Intl. Conf. on World Wide Web*, pages 293–304.

Kamra, A., Misra, V., and Nahum, E. (2004). Yaksha: A controller for managing the performance of 3-tiered websites. In *Proceedings of the 12th IWQoS*.

Kangasharju, J., Roberts, J., and Ross, K. (2001a). Object Replication Strategies in Content Distribution Networks. In *6th Web Caching Workshop*, Amsterdam. North-Holland.

Kangasharju, J., Ross, K., and Roberts, J. (2001b). Performance Evaluation of Redirection Schemes in Content Distribution Networks. *Computer Communications*, 24(2).

Karaul, M., Korilis, Y., and Orda, A. (1998). A Market-Based Architecture for Management of Geographically Dispersed, Replicated Web Servers. In *1st International Conference on Information and Computation Economics*, pages 158–165, New York, NY. ACM, ACM Press.

Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., and Yerushalmi, Y. (1999). Web caching with consistent hashing. In *Proc. 8th Intl. Conf. on World Wide Web*.

Karlsson, M., Karamanolis, C., and Mahalingam, M. (2002). A Framework for Evaluating Replica Placement Algorithms. Technical report, HP Laboratories, Palo Alto, CA.

Kemme, B. and Alonso, G. (1998). A suite of database replication protocols based on group communication primitives. In *Proc. ICDCS*, Washington, DC, USA.

Kemme, B. and Alonso, G. (2000). Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 134–143, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Krishnakumar, N. and Bernstein, A. J. (1994). Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems*, 4(19):586–625.

Krishnamurthy, B. and Wang, J. (2000). On Network-Aware Clustering of Web Clients. In *SIGCOMM*, pages 97–110, New York, NY. ACM, ACM Press.

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. (2000). Oceanstore: an architecture for global-scale persistent storage. In *Proc. 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 190–201.

Lai, K. and Baker, M. (1999). Measuring Bandwidth. In *18th INFOCOM Conference*, pages 235–245, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Larson, P., Goldstein, J., Guo, H., and Zhou, J. (2004). MTCache: Mid-tier database caching for SQL server. *Data Engineering*, 27(2):27–33.

Lavenberg, S. and Reiser, M. (1980). Stationary state probabilities at arrival instants for closed queueing networks with multiple types of customers. *Journal Applied Probability*, 17(4):1048–1061.

Leighton, F. and Lewin, D. (2000). Global Hosting System. United States Patent, Number 6,108,703.

Li, B., Golin, M. J., Italiano, G. F., and Deng, X. (1999). On the Optimal Placement of Web Proxies in the Internet. In *18th INFOCOM Conference*, pages 1282–1290, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Li, W.-S., Po, O., Hsiung, W.-P., Candan, K. S., and Agrawal, D. (2003). Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proceedings of the Twelfth international conference on World Wide Web*, pages 587–598. ACM Press.

Lin, Y., Kemme, B., Patiño-Martínez, M., and Jiménez-Peris, R. (2005). Middleware based data replication providing snapshot isolation. In *SIGMOD Conference*.

Luo, Q. and Naughton, J. F. (2001). Form-based proxy caching for database-backed web sites. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 191–200, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Manjhi, A., Gibbons, P. B., Ailamaki, A., Garrod, C., Maggs, B. M., Mowry, T. C., Olston, C., Tomasic, A., and Yu, H. (2007). Invalidation clues for database scalability services. In *Proceedings of ICDE*.

Mao, Z., Cranor, C., Douglis, F., Rabinovich, M., Spatscheck, O., and Wang, J. (2002). A precise and efficient evaluation of the proximity between web clients and their local dns servers.

McCune, T. and Andresen, D. (1998). Towards a Hierarchical Scheduling System for Distributed WWW Server Clusters. In *7th International Symposium on High Performance Distributed Computing*, pages 301–309, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

McManus, P. R. (1999). A Passive System for Server Selection within Mirrored Resource Environments Using AS Path Length Heuristics.

Mei, R. v. d., Gijsen, B., Engelberts, P., Berg, J. v. d., and Wingerden, K. v. (2006). Response times in queueing networks with feedback. *Performance Evaluation*, 64.

Mei, R. v. d. and Meeuwissen, H. (2006). Modelling end-to-end quality-of-service for transaction-based services in a multi-domain environment. In *Proceedings IEEE International Conference on Web Services ICWS*, Chicago, USA.

Menasce, D. A. (2002). Tpc-w: A benchmark for e-commerce. *IEEE Internet Computing*, 06(3):83–87.

Menasce, D. A. (2003). Web server software architectures. *IEEE Internet Computing*, 7(6):78–81.

Mockapetris, P. (1987a). Domain Names - Concepts and Facilities. RFC 1034.

Mockapetris, P. (1987b). Domain Names - Implementation and Specification. RFC 1035.

Mogul, J. C., Douglis, F., Feldmann, A., and Krishnamurthy, B. (1997). Potential Benefits of Delta Encoding and Data Compression for HTTP. In *SIGCOMM*, pages 181–194, New York, NY. ACM, ACM Press.

Moore, K., Cox, J., and Green, S. (1996). Sonar - A network proximity service. Internet-draft. [Online] http://www.netlib.org/utk/projects/sonar/.

Morrison, J. (1985). Response-time distribution for a processor-sharing system. *SIAM Journal on Applied Mathematics*, 45(1):152–167.

Mosberger, D. (1993). Memory Consistency Models. *Operating Systems Review*, 27(1):18–26.

Naik, V. K., Sivasubramanian, S., and Krishnan, S. (2004). Adaptive resource sharing in a web services environment. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 311–330, New York, NY, USA. Springer-Verlag New York, Inc.

Nelder, J. A. and Mead, R. (1965). A Simplex Method for Function Minimization. *The Computer Journal*, 4(7).

Ng, E. and Zhang, H. (2002). Predicting Internet Network Distance with Coordinates-Based Approaches. In *21st INFOCOM Conference*, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Ninan, A., Kulkarni, P., Shenoy, P., Ramamritham, K., and Tewari, R. (2002). Cooperative Leases: Scalable Consistency Maintenance in Content Distribution Networks. In *11th International World Wide Web Conference*, pages 1–12, New York, NY. ACM Press.

Obraczka, K. and Silva, F. (2000). Network Latency Metrics for Server Proximity. In *Globecom*, San Francisco, CA. IEEE.

Odlyzko, A. (2001). Internet Pricing and the History of Communications. *Computer Networks*, 36:493–517.

Olston, C., Manjhi, A., Garrod, C., Ailamaki, A., Maggs, B., and Mowry, T. (2005). A scalability service for dynamic web applications. In *Proc. CIDR Conf.*, pages 56–69.

Ott, T. (1984). The sojourn time distribution in the M/G/1 queue with processor sharing. *Journal of Applied Probability*, 21:360–378.

Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. (1998). Locality-Aware Request Distribution in Cluster-Based Network Servers. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, New York, NY. ACM, ACM Press.

Pansiot, J. and Grad, D. (1998). On Routes and Multicast Trees in the Internet. *ACM Computer Communications Review*, 28(1):41–50.

Paxson, V. (1997a). End-to-end routing behavior in the internet. *IEEE/ACM Trans. Netw.*, 5(5):601–615.

Paxson, V. (1997b). Measurements and Analysis of End-to-End Internet Dynamics. Technical Report UCB/CSD-97-945, University of California at Berkeley.

Pias, M., Crowcroft, J., Wilbur, S., Harris, T., and Bhatti, S. (2003). Lighthouses for Scalable Distributed Location. In *2nd International Workshop on Peer-to-Peer Systems*, Berlin. Springer-Verlag.

Pierre, G. and van Steen, M. (2006). Globule: a collaborative content delivery network. *IEEE Communications Magazine*, 44(8):127–133.

Pierre, G., van Steen, M., and Tanenbaum, A. (2002). Dynamically selecting optimal distribution strategies for Web documents. *IEEE Transactions on Computers*, 51(6):637–651.

Plattner, C. and Alonso, G. (2004). Ganymed: scalable replication for transactional web applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, New York, NY, USA. Springer-Verlag New York, Inc.

Plattner, C., Alonso, G., and zsu, M. T. (2006). Dbfarm: A scalable cluster for multiple databases. In *Middleware '06: Proceedings of the 7th ACM/IFIP/USENIX international conference on Middleware*, New York, NY, USA. Springer-Verlag New York, Inc.

Popescu, B. C., van Steen, M., Crispo, B., Tanenbaum, A. S., Sacha, J., and Kuz, I. (2005). Securely replicated web documents. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 104.2, Washington, DC, USA. IEEE Computer Society.

Qiu, L., Padmanabhan, V., and Voelker, G. (2001). On the Placement of Web Server Replicas. In *20th INFOCOM Conference*, pages 1587–1596, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Rabinovich, M. and Aggarwal, A. (1999). Radar: A Scalable Architecture for a Global Web Hosting Service. *Computer Networks*, 31(11–16):1545–1561.

Rabinovich, M. and Spastscheck, O. (2002). *Web Caching and Replication*. Addison-Wesley, Reading, MA.

Rabinovich, M., Triukose, S., Wen, Z., , and Wang., L. (2006). Dipzoom: The internet measurements marketplace. the 9th ieee global internet symp., may 2006.

Rabinovich, M., Xiao, Z., and Agarwal, A. (2003). Computing on the edge: A platform for replicating internet applications. In *Proc. of the Eighth International Workshop on Web Content Caching and Distribution*, Hawthorne, NY, USA.

Rabinovich, M., Xiao, Z., Douglis, F., and Kalmanek, C. (1997). Moving edge-side includes to the real edge- the clients. In *USENIX Symposium on Internet Technologies and Systems*.

Radoslavov, P., Govindan, R., and Estrin, D. (2001). Topology-Informed Internet Replica Placement. In *6th Web Caching Workshop*, Amsterdam. North-Holland.

Radware (2002). Web Server Director. Technical report, Radware, Inc.

Raynal, M. and Singhal, M. (1996). Logical Time: Capturing Causality in Distributed Systems. *Computer*, 29(2):49–56.

Rekhter, Y. and Li, T. (1995). A Border Gateway Protocol 4 (BGP-4). RFC 1771.

Rilling, L., Sivasubramanian, S., and Pierre, G. (2007). High availability and scalability support for web applications. In *Proceedings of the IEEE International Symposium on Applications and the Internet*, Hiroshima, Japan.

Rodriguez, P., Kirpal, A., and Biersack, E. (2000). Parallel-Access for Mirror Sites in the Internet. In *19th INFOCOM Conference*, pages 864–873, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Rodriguez, P. and Sibal, S. (2000). SPREAD: Scalable Platform for Reliable and Efficient Automated Distribution. *Computer Networks*, 33(1–6):33–46.

Rodriguez, P., Spanner, C., and Biersack, E. (2001). Analysis of Web Caching Architecture: Hierarchical and Distributed Caching. *IEEE/ACM Transactions on Networking*, 21(4):404–418.

Rosen, E., Viswanathan, A., and Callon, R. (2001). Multiprotocol label switching architecture.

Sayal, M., Sheuermann, P., and Vingralek, R. (2003). Content Replication in Web++. In *2nd International Symposium on Network Computing and Applications*, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Seltzsam, S., Gmach, D., Krompass, S., and Kemper, A. (2006). Autoglobe: An automatic administration concept for service-oriented database applications. In *Proc. 22nd Intl. Conf. on Data Engineering*.

Shaikh, A., Tewari, R., and Agrawal, M. (2001). On the Effectiveness of DNS-based Server Selection. In *20th INFOCOM Conference*, pages 1801–1810, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Shavitt, Y. and Tankel, T. (2003). Big-Bang Simulation for Embedding Network Distances in Euclidean Space. In *22nd IEEE INFOCOM*, San Francisco, CA, USA.

Shmoys, D., Tardos, E., and Aardal, K. (1997). Approximation Algorithms for Facility Location Problems. In *29th Symposium on Theory of Computing*, pages 265–274, New York, NY. ACM, ACM Press.

Shneiderman, B. (1984). Response time and display rate in human performance with computers. *ACM Computing Surveys*, 16(3):265–285.

Sivasubramanian, S., Alonso, G., Pierre, G., and van Steen, M. (2005). Globedb: autonomic data replication for web applications. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 33–42, New York, NY, USA. ACM Press.

Sivasubramanian, S., Pierre, G., and van Steen, M. (2003). A case for dynamic selection of replication and caching strategies. In *Proceedings of the Eighth International Workshop Web Content Caching and Distribution*, pages 275–282, Hawthorne, NY, USA.

Sivasubramanian, S., Pierre, G., and van Steen, M. (2004a). Scalable strong consistency for web applications. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, page 33, New York, NY, USA. ACM Press.

Sivasubramanian, S., Pierre, G., and van Steen, M. (2006a). Towards autonomic hosting of multi-tier internet applications. In *Proceedings of the HotAC-I Workshop*.

Sivasubramanian, S., Pierre, G., van Steen, M., and Alonso, G. (2006b). GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands. http://www.globule.org/publi/GCBRCDWA_ircs022.html.

Sivasubramanian, S., Szymaniak, M., Pierre, G., and van Steen, M. (2004b). Replication for web hosting systems. *ACM Computing Surveys*, 36(3):291–334.

Soundararajan, G., Amza, C., and Goel, A. (2006). Database replication policies for dynamic content applications. In *Proceedings of 1st ACM EuroSys conference*.

Stemm, M., Katz, R., and Seshan, S. (2000). A Network Measurement Architecture for Adaptive Applications. In *19th INFOCOM Conference*, pages 285–294, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Su, A.-J., Choffnes, D. R., Kuzmanovic, A., and Bustamante, F. E. (2006). Drafting behind akamai (travelocity-based detouring). *SIGCOMM Comput. Commun. Rev.*, 36(4):435–446.

Szymaniak, M., Pierre, G., and van Steen, M. (2003). NetAirt: A Flexible Redirection System for Apache. In *Proceedings of the IADIS International Conference on WWW/Internet*, Algarve, Portugal.

Szymaniak, M., Pierre, G., and van Steen, M. (2004). Scalable Cooperative Latency Estimation. In *Proceedings of the 10th International Conference on Parallel and Distributed Systems (ICPADS)*, Newport Beach, CA, USA.

Szymaniak, M., Pierre, G., and van Steen, M. (2006). Latency-driven replica placement. *IPSJ Journal*, 47(8). http://www.globule.org/publi/LDRP_ipsj2006.html.

Tang, X. and Xu, J. (2004). On replica placement for qos-aware content distribution.

Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M., and Welsh, B. (1994). Session Guarantees for Weakly Consistent Replicated Data. In *3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Tewari, R., Niranjan, T., and Ramamurthy, S. (2002). WCDP: A Protocol for Web Cache Consistency. In *7th Web Caching Workshop*.

Tijms, H. (1994). *Stochastic Models: An Algorithmic Approach*. John Wiley & Sons.

Torres-Rojas, F. J., Ahamad, M., and Raynal, M. (1999). Timed consistency for shared distributed objects. In *18th Symposium on Principles of Distributed Computing*, pages 163–172, New York, NY. ACM, ACM Press.

Trivedi, K. S. (2002). *Probability and statistics with reliability, queuing and computer science applications*. John Wiley and Sons Ltd., Chichester, UK, UK.

Ullman, J. D. (1990). *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA.

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2005). An analytical model for multi-tier internet services and its applications. In *Proc. SIGMETRICS Intl. Conf.*, pages 291–302.

Urgaonkar, B. and Shenoy, P. (2005). Cataclysm: policing extreme overloads in internet applications. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 740–749, New York, NY, USA. ACM Press.

Verma, D. C. (2002). *Content Distribution Networks: An Engineering Approach*. John Wiley, New York.

Villela, D., Pradhan, P., and Rubenstein, D. (2004). Provisioning servers in the application tier for e-commerce systems. In *Proc. 12th IEEE Intl. Workshop on Quality of Service*.

Vogels, W. (2006). Learning from the amazon technology platform. *ACM Queue*, 4(4).

Waldvogel, M. and Rinaldi, R. (2003). Efficient topology-aware overlay network. *ACM Computer Communications Review*, 33(1):101–106.

Wang, J. (1999). A Survey of Web Caching Schemes for the Internet. *ACM Computer Communications Review*, 29(5):36–46.

Wang, L., Pai, V., and Peterson, L. (2002). The Effectiveness of Request Redirection on CDN Robustness. In *5th Symposium on Operating System Design and Implementation*, Berkeley, CA. USENIX, USENIX.

Wolski, R., Spring, N., and Hayes, J. (1999). The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768.

Xiao, J. and Zhang, Y. (2001). Clustering of Web Users Using Session-Based Similarity Measures. In *International Conference on Computer Networks and Mobile Computing*, pages 223–228, Los Alamitos, CA. IEEE, IEEE Computer Society Press.

Yang, J. (2005). Data clustering for autonomic application replication. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands.

Yin, J., Alvisi, L., Dahlin, M., and Iyengar, A. (2002). Engineering Web Cache Consistency. *ACM Transactions on Internet Technology*, 2(3):224–259.

Yu, H. and Vahdat, A. (2000). Efficient Numerical Error Bounding for Replicated Network Services. In Abbadi, A. E., Brodie, M. L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., and Whang, K.-Y., editors, *26th International Conference on Very Large Data Bases*, pages 123–133, San Mateo, CA. Morgan Kaufman.

Yu, H. and Vahdat, A. (2002). Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *ACM Transactions on Computer Systems*, 20(3):239–282.

Zari, M., Saiedian, H., and Naeem, M. (2001). Understanding and Reducing Web Delays. *Computer*, 34(12):30–37.

Zhao, B., Huang, L., Stribling, J., Rhea, S., Joseph, A., and Kubiatowicz, J. (2004). Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communication*, 22(1):41–53.